University of Alberta

FAULT TOLERANCE THROUGH REDUNDANCY FOR BATCH WORKLOADS

by

Paul Nalos

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2006

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-22331-4
Our file Notre référence
ISBN: 978-0-494-22331-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Many applications in computational science can be expressed as *batch workloads*, sets of jobs that communicate via a file system. Users are often concerned with *makespan*, the total time required to complete a workload. During workload execution, failures or slowness in one resource may prolong makespan while many alternative resources are idle: the *long tail* problem.

We address the long tail problem by extending the Trellis metacomputing system to redundantly execute jobs while isolating redundant executions from each other. For example, a job may be redundantly executed on two nodes, both independently computing the job's output, but only one node's copy of this output is saved.

Our system maintains high performance in the face of slow or failed nodes, even in cases where manual intervention was previously required to make forward progress. We evaluate our system with workloads from Proteome Analyst, a bioinformatics application developed at the University of Alberta.

Acknowledgements

I am grateful to the many people who have, directly or indirectly, contributed to this work. My supervisor, Paul Lu, has played a key role mentoring me to recognize, critique, and produce quality research. His insightful guidance has significantly shaped this thesis. The Trellis project, which this work builds upon, is the result of years of hard work by a large group of people. In particular, I would like to thank Michael Closson, Cam Macdonell, and Yang Wang, whom I worked with closely. This work also builds on the Proteome Analyst application; I appreciate the Proteome Analyst Research Group's willingness to allow me to participate in their meetings as they search for innovative ways to automatically annotate proteins and proteomes. For a course project, Nicholas Boers, Alona Fyshe, Adam Rachmielowski, and I built on work by Nicholas Lamb, extending Proteome Analyst to execute the HMMER tools in a distributed fashion with Trellis [4]. The experimental evaluation is exclusively based on the workloads made possible by that project, and it has been a great benefit to this thesis. Finally, I thank my wife, Rachel, whose love and support has made this both possible and fun.

To Rachel

Table of Contents

1 Introduction 1
- 1.1 Contributions . . . 5

2 Background 7
- 2.1 Background Concepts . . . 7
  - 2.1.1 The Trellis System . . . 7
  - 2.1.2 Typical Workloads . . . 8
  - 2.1.3 The Long Tail Problem . . . 8
  - 2.1.4 Faults and Redundancy . . . 11
- 2.2 Related Work . . . 12
  - 2.2.1 Charlotte . . . 12
  - 2.2.2 Satin . . . 12
  - 2.2.3 BAD-FS . . . 13
  - 2.2.4 MapReduce . . . 14
  - 2.2.5 Transactional File Systems . . . 15
- 2.3 Concluding Remarks . . . 15

3 Architecture 17
- 3.1 Three Deployment Strategies for PA/Trellis . . . 17
  - 3.1.1 Proteome Analyst . . . 17
  - 3.1.2 Trellis Driver . . . 21
  - 3.1.3 Placeholder Scheduling System . . . 21
  - 3.1.4 HMMER . . . 22
  - 3.1.5 NFS . . . 22
  - 3.1.6 Trellis File System . . . 22
  - 3.1.7 Trellis Samba . . . 23
- 3.2 A Framework for Multiple Write-back Policies in the Trellis File System Library . 23
  - 3.2.1 Deviation from POSIX Semantics . . . 23
  - 3.2.2 Interaction with the New Cache Manager . . . 24
  - 3.2.3 Write-back Policies . . . 25
- 3.3 Scheduling for Isolated, Redundant Execution . . . 26
  - 3.3.1 Job Selection Policy . . . 28
  - 3.3.2 Scheduler and File System Integration . . . 28
- 3.4 Concluding Remarks . . . 29

4 Implementation 31
- 4.1 Extensions to the Trellis File System . . . 31
- 4.2 Extensions to the Placeholder Scheduling System . . . 31
  - 4.2.1 Database Schema . . . 31
  - 4.2.2 Placeholder-facing Interface . . . 33
  - 4.2.3 User Interface . . . 34
- 4.3 TFS Wrapper . . . 34
- 4.4 Concluding Remarks . . . 36

5 Empirical Evaluation 37
- 5.1 Evaluation Methodology . . . 37
- 5.2 Experimental Platform . . . 42
- 5.3 Fault Tolerance and Multiple Administrative Domains . . . 45
- 5.4 Slow Nodes . . . 45
- 5.5 Overheads . . . 46
  - 5.5.1 HMMER via hmmpfam (Medium-Grain Jobs) . . . 54

5.5.2 HMMER via hmmsearch (Fine-Grain Jobs) 54
5.5.3 Secure-shell Connection Sharing 64
5.5.4 Batching 64
5.6 Future Work: Alternative Workloads 65
5.7 Concluding Remarks 68

6 Conclusion **72**

Bibliography **74**

A Acronyms and Programs **77**
A.1 List of Acronyms 77
A.2 List of Programs 78

List of Tables

3.1 Write-back Policies 25

4.1 Cache Manager Interface 33
4.2 Placeholder Interface: Request Job 34
4.3 Placeholder Interface: Job Done 35
4.4 Placeholder Interface: Renew Lease 35
4.5 Placeholder Interface: Commit Done 35

5.1 High-level Overheads 39
5.2 Placeholder Overheads Part I: Overheads Based on the Placeholder Logs 40
5.3 Placeholder Overheads Part II: Overheads Based on the Job Database 41
5.4 File-system Access Mechanisms 44
5.5 Experimental Platform A – Three Nodes in Two Administrative Domains 44
5.6 Experimental Platform B – Two Heterogeneous Nodes with Samba 44
5.7 Experimental Platform C – Cluster 44

List of Figures

1.1 The Impact of Redundant Execution on Fault Tolerance . . . 3
1.2 The Impact of Redundant Execution with Slow Nodes . . . 4

2.1 Sample Trellis Metacomputer Topology . . . 9
2.2 Hypothetical Proteome Analyst Workload Continued: Jobs Access Input and Output Data via the File System . . . 10

3.1 Architecture of PA/Trellis with NFS . . . 18
3.2 Architecture of PA/Trellis with the Trellis File System Library . . . 19
3.3 Architecture of PA/Trellis with Trellis Samba . . . 20
3.4 Code Fragment: Submitting Work to a Trellis Metacomputer with Trellis Driver . 22
3.5 Job State Transitions . . . 27

4.1 Scheduler Database Schema (original) . . . 32

5.1 Execution Time Breakdown – PA Analyzes a Proteome . . . 38
5.2 Using HMMER to Match Proteins to Protein Families: hmmpfam versus hmmsearch 43
5.3 Impact of Redundant Execution with Slow Nodes: hmmpfam, 343 proteins, 4×2 cluster CPUs . . . 46
5.4 Performance Breakdown: hmmpfam, 343 proteins, 4×2 cluster CPUs . . . 47
5.5 Placeholder Overheads: hmmpfam, 343 proteins, 4×2 cluster CPUs . . . 48
5.6 Performance Breakdown: hmmpfam, 343 proteins, 4×2 cluster CPUs, one node slowed 5x . . . 49
5.7 Placeholder Overheads: hmmpfam, 343 proteins, 4×2 cluster CPUs, one node slowed 5x . . . 50
5.8 Performance Breakdown: hmmpfam, 343 proteins, 4×2 cluster CPUs, one node slowed 10x . . . 51
5.9 Placeholder Overheads: hmmpfam, 343 proteins, 4×2 cluster CPUs, one node slowed 10x . . . 52
5.10 Relative Performance: hmmpfam versus hmmsearch, 343 proteins . . . 53
5.11 Performance Breakdown: hmmpfam, 343 proteins, two heterogeneous nodes . . . 55
5.12 Placeholder Overheads: hmmpfam, 343 proteins, two heterogeneous nodes . . . 56
5.13 Performance Breakdown: hmmpfam, 343 proteins, 8×2 cluster CPUs . . . 57
5.14 Placeholder Overheads: hmmpfam, 343 proteins, 8×2 cluster CPUs . . . 58
5.15 Performance Breakdown: hmmsearch, 343 proteins, two heterogeneous nodes . . . 60
5.16 Placeholder Overheads: hmmsearch, 343 proteins, two heterogeneous nodes . . . 61
5.17 Performance Breakdown: hmmsearch, 343 proteins, 4×2 cluster CPUs . . . 62
5.18 Placeholder Overheads: hmmsearch, 343 proteins, 4×2 cluster CPUs . . . 63
5.19 Impact of Secure-shell Connection Sharing: hmmsearch, 343 proteins, 4×2 cluster CPUs . . . 65
5.20 Performance Breakdown: hmmsearch, 343 proteins, 4×2 cluster CPUs, without secure-shell connection sharing . . . 66
5.21 Placeholder Overheads: hmmsearch, 343 proteins, 4×2 cluster CPUs, without secure-shell connection sharing . . . 67
5.22 Impact of Batching: hmmsearch, 343 proteins, 4×2 cluster CPUs . . . 68
5.23 Performance Breakdown: hmmsearch, 343 proteins, 4×2 cluster CPUs, batching factor 32 . . . 69
5.24 Placeholder Overheads: hmmsearch, 343 proteins, 4×2 cluster CPUs, batching factor 32 . . . 70

Chapter 1

Introduction

Computational science is the use of computers to solve scientific problems. One natural division in computational science arises between capability and capacity computing [28]. *Capability computing* is focused on systems that can solve problems other systems cannot. In pursuit of this goal, such systems might include many processors and large memories connected by a high-performance interconnect. In contrast, *capacity computing* is focused on solving problems by aggregating large numbers of general purpose computing resources. It is reasonable for these resources to be heterogeneous and loosely connected across a wide area. An important subset of problems in computational science can be efficiently addressed by capacity computing systems. For example, parameter studies, where the same computation is independently repeated many times with different parameter choices, are a natural fit. The capability / capacity distinction is not entirely black and white; in hybrid approaches, collections of highly capable systems cooperate over a wide area.

Trellis [31] is a collection of software which allows users to create *overlay metacomputers*, aggregations of computing resources that function cohesively as a single system. As a capacity computing system, Trellis is able to leverage heterogeneous computing resources across a wide area; a Trellis metacomputer can be constructed entirely at user level and include computers from multiple administrative domains. Work is submitted to Trellis in the form of jobs which are scheduled concurrently onto different systems while respecting dependencies. Jobs typically access their input and output data via a distributed file system. Computational resources can be dynamically added and removed while the metacomputer is executing jobs. Trellis was originally created to explore alternative choices in the metacomputing design space. A key goal was to allow unprivileged users to aggregate computing resources without the need for administrative support or special infrastructure. The Trellis project is working towards the goal of supporting arbitrary applications across multiple administrative domains [15]. In a series of experiments known as the *Canadian Internetworked Scientific Supercomputer* (CISS; see Appendix A.1 for a list acronyms), Trellis has been used to solve large problems in chemistry, physics, and molecular dynamics [20, 34, 14].

Without the extensions described in this work, a Trellis metacomputer's performance is very sensitive to performance variations and failures in its constituent nodes. For example, if one of

the last jobs in a workload is assigned to a node that subsequently fails, the scheduler will wait indefinitely for the failed node to complete the job, even if there are idle nodes that could complete it as well. Only after the user intervenes will the job be reassigned, allowing the workload to complete. Similarly, if this job is assigned to a node that executes very slowly, the scheduler will wait for the slow execution to complete, even if another node becomes available that could complete the job sooner. In both cases, makespan (the total time to complete a workload) is drawn out while many nodes are idle. When this happens, we say the workload has a *long tail*. In contrast, if one of many independent jobs is delayed at the beginning of a workload, the impact of the delay is minimal as long as the job completes before other nodes run out of work.

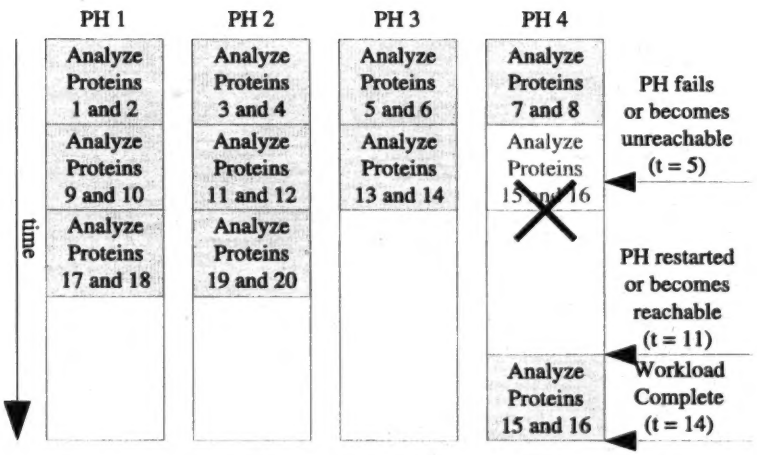
Failures and performance variation are the norm in large distributed systems. Computers and networks can fail, become temporarily unavailable, or suffer from degraded performance. As more and more resources are aggregated, the overall exposure to failure increases, as there are more components to fail. To a certain extent it is possible to identify and eliminate slow nodes and those with unpredictable performance or stability, but this is undesirable for two reasons. First, failures or performance degradation can occur suddenly and unpredictably on otherwise healthy nodes for a variety of reasons, including resource contention, misconfiguration, or failing hard disks [7, 22]. Second, even slow or unstable nodes – when functional – can make meaningful contributions to completing an overall workload.

To mitigate the effect of failures on the workload tail, we extend the Trellis scheduler [19, 18] to execute jobs redundantly when there are idle nodes. In this way, failed or pathologically slow job executions are eventually bypassed by successful executions on other nodes, ultimately resulting in reduced makespan.

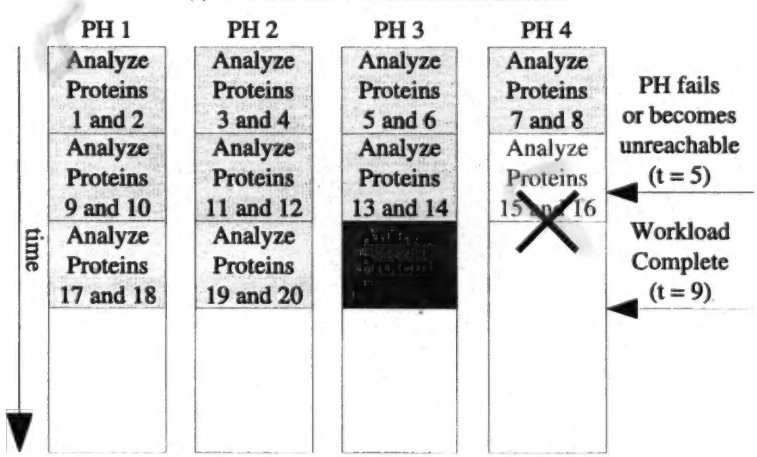
In the following illustrative examples we use a hypothetical workload from Proteome Analyst [29] (a bioinformatics application developed at the University of Alberta) consisting of ten independent jobs. Each job analyzes two protein sequences to determine some of their biological properties. This workload is executed on a Trellis metacomputer with four CPUs (e.g., four nodes with one CPU each). A *placeholder* runs on each of the CPUs and pulls work from the scheduler to execute locally. We are interested in two cases:

1. **Node or network failure.** Figure 1.1 illustrates how Trellis behaves, with and without our extensions, when a placeholder that is running a job stops communicating with the scheduler. Typically, this happens because the node hosting the placeholder fails or because the network between the scheduler and placeholder becomes unavailable. Without redundant execution, the workload cannot be completed until the failure is resolved. In the case of node failure, resolution requires user intervention. With redundant execution, the workload completes before the failure is resolved because the stranded job is redundantly executed on another placeholder.

2. **Slow job executions.** Figure 1.2 demonstrates the impact of redundant execution when one

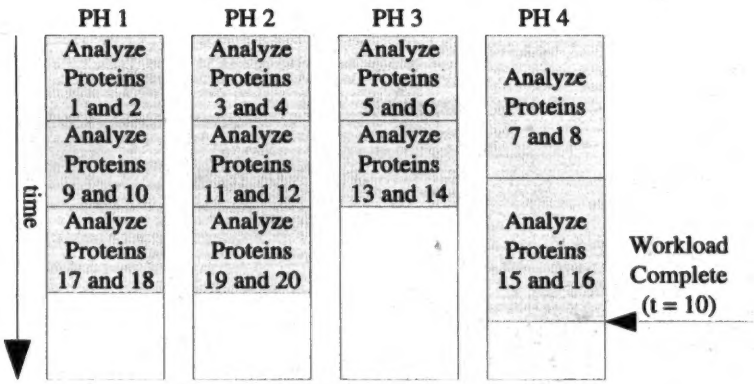

PH 1
PH 2
PH 3
PH 4
Analyze Proteins 1 and 2
Analyze Proteins 3 and 4
Analyze Proteins 5 and 6
Analyze Proteins 7 and 8
Analyze Proteins 9 and 10
Analyze Proteins 11 and 12
Analyze Proteins 13 and 14
Analyze Proteins 15 and 16
Analyze Proteins 17 and 18
Analyze Proteins 19 and 20
time
PH fails or becomes unreachable (t = 5)
PH restarted or becomes reachable (t = 11)
Workload Complete (t = 14)

(a) Fault Tolerance without Redundant Execution

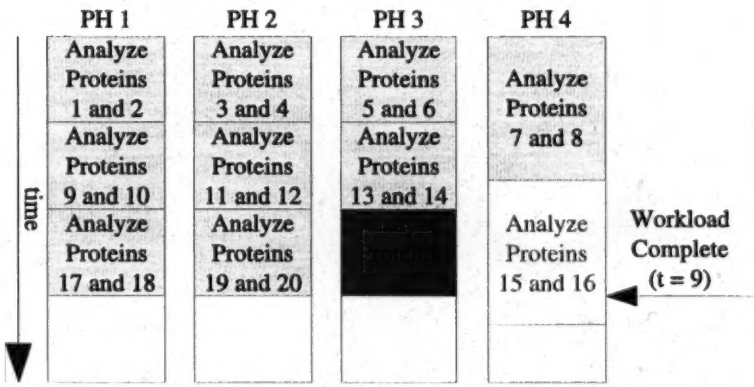

PH 1
PH 2
PH 3
PH 4
Analyze Proteins 1 and 2
Analyze Proteins 3 and 4
Analyze Proteins 5 and 6
Analyze Proteins 7 and 8
Analyze Proteins 9 and 10
Analyze Proteins 11 and 12
Analyze Proteins 13 and 14
Analyze Proteins 15 and 16
Analyze Proteins 17 and 18
Analyze Proteins 19 and 20
time
PH fails or becomes unreachable (t = 5)
Workload Complete (t = 9)

(b) Fault Tolerance with Redundant Execution. The redundant execution is shown in dark gray.

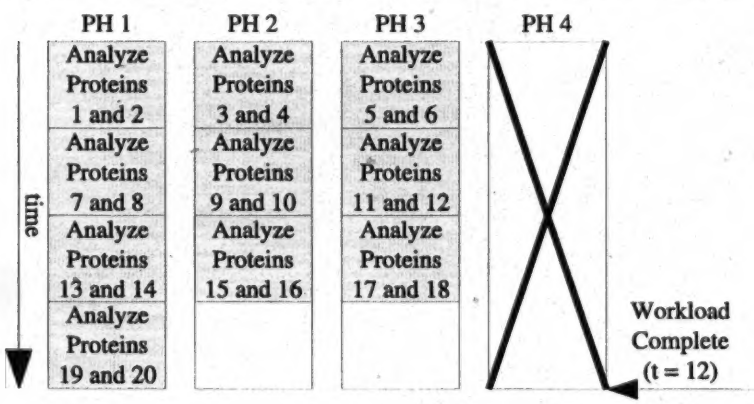
Figure 1.1: **The Impact of Redundant Execution on Fault Tolerance.** Example job schedules are shown with and without redundant execution. Each job takes three time units to complete, during which two proteins are analyzed. Four placeholders collectively provide a maximum degree of concurrency of four. At time $t = 5$, Placeholder PH 4 becomes unreachable due to a network interruption or node failure. At time $t = 11$, the failure is resolved; in the case of node failure, this requires manual intervention. In this example, redundant execution reduces makespan from 11 time units to 9 time units.


PH 1
PH 2
PH 3
PH 4
Analyze Proteins 1 and 2
Analyze Proteins 3 and 4
Analyze Proteins 5 and 6
Analyze Proteins 7 and 8
Analyze Proteins 9 and 10
Analyze Proteins 11 and 12
Analyze Proteins 13 and 14
Analyze Proteins 17 and 18
Analyze Proteins 19 and 20
Analyze Proteins 15 and 16
time
Workload Complete (t = 10)

(a) Slow Nodes without Redundant Execution


PH 1
PH 2
PH 3
PH 4
Analyze Proteins 1 and 2
Analyze Proteins 3 and 4
Analyze Proteins 5 and 6
Analyze Proteins 7 and 8
Analyze Proteins 9 and 10
Analyze Proteins 11 and 12
Analyze Proteins 13 and 14
Analyze Proteins 15 and 16
Analyze Proteins 17 and 18
Analyze Proteins 19 and 20
time
Workload Complete (t = 9)

(b) Slow Nodes with Redundant Execution. The redundant execution is shown in dark gray.


PH 1
PH 2
PH 3
PH 4
Analyze Proteins 1 and 2
Analyze Proteins 3 and 4
Analyze Proteins 5 and 6
Analyze Proteins 7 and 8
Analyze Proteins 9 and 10
Analyze Proteins 11 and 12
Analyze Proteins 13 and 14
Analyze Proteins 15 and 16
Analyze Proteins 17 and 18
Analyze Proteins 19 and 20
time
Workload Complete (t = 12)

(c) Slow Nodes Unused

Figure 1.2: **The Impact of Redundant Execution with Slow Nodes.** Here, Placeholder PH 4 takes 5 time units to complete jobs; the other placeholders complete jobs in only 3 time units. Example job schedules are shown (a) without redundant execution, (b) with redundant execution, and (c) with Placeholder PH 4 left unused. Without redundant execution, the workload completes in 10 time units. Redundant execution reduces this to 9 time units, while removing the slow placeholder increases makespan to 12 time units. Slow placeholders provide some benefit during the main part of the workload, but without redundant execution can be a liability at the end of the workload.

of the four placeholders is slower than the others. Redundant execution reduces makespan by bypassing a slow execution at the end of the workload. In contrast, removing the slow placeholder from the metacomputer altogether (Figure 1.2c) increases makespan. This is because the slow placeholder is a benefit during the main part of the workload, and a liability at the end; redundant execution mitigates the liability without compromising the benefit.

One significant challenge for redundant execution is interaction between job executions. Jobs access input, output, and intermediate data via a file system, and redundant executions may result in failures or corrupt output. For example, imagine that Job A reads from the file `A.in`, creates a directory called `A.out`, and writes the files `A.out/1` and `A.out/2`. Two concurrent executions of Job A may interfere with each other. Depending on implementation, perhaps only one execution will be able to create the output directory `A.out`, or perhaps one execution will write `A.out/1` while the other writes an inconsistent `A.out/2`. To overcome this challenge, there is a need for redundant executions of the same job to be *isolated* from each other. Specifically, redundant executions of the same job must not be able to observe file-system modifications made by each other.

To provide the necessary isolation, we extend the Trellis File System [25] and scheduler [19, 18] to isolate file-system changes made during job execution. While a job executes, file-system changes are not immediately *persisted* (i.e., saved to disk so that the user and other jobs can observe them) but rather cached locally. After a job execution completes, all of its changes are either persisted together (if it was the first successful execution for the job) or discarded. In this way, redundant executions of jobs are prevented from observing each other's file-system changes. The integration between file system and scheduler is lightweight; the scheduler notifies the file system when each job begins, associates all file-system requests with specific jobs, and notifies the file system when to apply or discard pending changes for a given job. A consequence of this approach is that for applications to benefit from the redundant execution feature, they must use the Trellis File System to access their input and output data.

1.1 Contributions

This thesis makes the following contributions:

1. The Trellis File System (TFS) library is extended to support multiple write-back policies, including the *explicit write-back* policy, which provides a transaction-style interface to the file system. When the explicit write-back policy is in use, metadata and data are aggressively cached, and all file-system modifications are buffered locally. After a series of file-system modifications have been made, these changes may be persisted together or abandoned.

   Further, a module for the Samba file server is created allowing unmodified applications to access the Trellis File System library indirectly via a CIFS file-system mount. This provides an alternative to modifying and relinking user applications to access the TFS library directly.

2. The Trellis scheduler is extended to redundantly execute jobs when the available concurrency exceeds the number of ready jobs. This mechanism is integrated with the file system, using the explicit write-back policy, such that redundant executions of jobs do not see each other's file-system modifications, and only one set of these modifications is persisted. While jobs are running, aggressive file-system caching is possible because the scheduler is aware of dependencies between jobs and does not concurrently schedule jobs with read/write or write/write conflicts for the same paths.

3. We provide a detailed, quantitative analysis of the impact of redundant execution on real workloads driven by Proteome Analyst. We show that for medium- or large-grain, CPU-intensive workloads, redundant execution introduces negligible overheads and allows the system to transparently recover from failed or slow placeholders. For fine-grain workloads, the scheduling system, the Trellis File System, and redundant execution all introduce significant overheads.

Chapter 2

Background

In the first chapter, we discussed how the overall performance of capacity computing systems can be vulnerable to performance variations and failures in their constituent resources, using the Trellis system as an example. We motivate redundant job execution as one way to address this challenge, proposing specific file-system and scheduling extensions to make this possible. In this chapter, we explore relevant concepts in more detail and survey related work.

2.1 Background Concepts

Here, we introduce background concepts that provide a basis for the architecture and implementation details to follow: the Trellis system, typical workloads, the long tail problem, and different classes of failure.

2.1.1 The Trellis System

A Trellis metacomputer is a distributed system that contains some or all of the following components. One possible topology is shown in Figure 2.1.

1. **Users.** Users and external systems submit jobs to the Trellis metacomputer and monitor the jobs' progress. For example, a job could be a request to execute the BLAST [1] program with a specific set of parameters and input and output files. Trellis Driver [13] is an example of external software that submits and monitors jobs on behalf of Java applications (e.g., Proteome Analyst).

2. **Command Line Server (CLS, scheduler) [19, 18].** The CLS is the central scheduler and repository for job information. It stores jobs, and the dependencies between jobs, in a database; together these form a directed acyclic graph [10]. Jobs are represented simply as command lines.

3. **Placeholders [19, 18].** Placeholders, software entities that run on available computational resources, request (pull) jobs from the scheduler, and notify the scheduler when these jobs

complete. The most simple form of a placeholder – the *zero-infrastructure* placeholder – is a script that loops indefinitely requesting jobs, executing them, and notifying the scheduler of the results. More complicated placeholders can work with batch scheduling systems, executing one job each time a placeholder reaches the top of a job queue.

4. **Home nodes [25].** Home nodes are backing file servers that store all data accessed via the Trellis File System. The TFS library can access homes nodes with a variety of protocols, including *secure shell*. Secure shell is a suite of tools (including `ssh`; see Appendix A.2 for a list of programs) and a protocol that allows for secure remote command invocation, file transfer, and interactive login.

5. **Distributed File System [25, 6, 14].** Programs executed by placeholders access data stored on home nodes in one of two ways. Either they link directly to the Trellis File System library (not shown), or they access *bridging file servers*, which internally use the TFS library to provide home-node access.

2.1.2 Typical Workloads

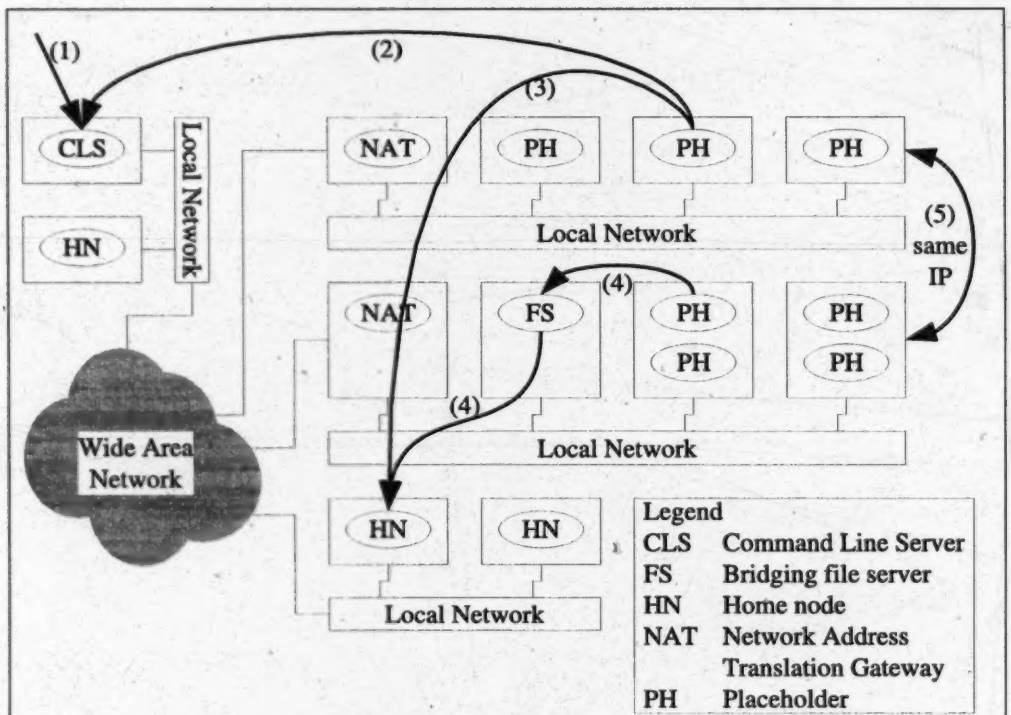
In Trellis, a *workload* is a set of jobs, potentially with interdependencies, that access input, output, and intermediate data via the file system. More generally, this type of workload is called a *batch workload*. In Trellis, *jobs* are simply command lines (i.e., invocations of existing programs). A wide variety of scientific computations can be expressed in this way [30].

In Chapter 1, we motivated the use of redundant execution by showing how a hypothetical Proteome Analyst workload is affected by different kinds of faults with and without redundant execution. Figure 2.2 shows this same workload from the perspective of file access. Jobs access all of their input and output data via the file system. Prior to submitting the workload, Proteome Analyst produces one small input file per protein, and after the workload completes, Proteome Analyst reads each of the corresponding output files.

2.1.3 The Long Tail Problem

In the scheduling literature, *makespan* is the duration of time between the submission of the first job in a workload and the completion of the last job in the workload. Makespan is relative to the system under discussion. For example, Proteome Analyst's makespan is measured from the moment a user submits a proteome for analysis to the moment results are available, even though PA might complete part of this task by submitting jobs to a Trellis metacomputer. For batch workloads, makespan is arguably the most important performance metric: "How long will my workload take to complete?"

A workload execution is *long tailed* if most placeholders run out of work long before the last few jobs complete. In the example from Figure 1.1a, as a result of a node or network failure, Placeholder PH 4 finishes the last job long (5 time units) after the other jobs have completed. A



1. Users connect to the CLS to submit jobs or query for status.
2. Placeholders connect to the CLS to request work and to report the ensuing success or failure.
3. Programs executed by placeholders connect to home nodes for file-system access.
4. As an alternate to (3), programs executed by placeholders access the distributed file system via bridging file servers within their local networks. In turn, these servers communicate with the home nodes on the programs' behalf.
5. Placeholders, which never need to be contacted, may reside on nodes without unique IP addresses.

Figure 2.1: **Sample Trellis Metacomputer Topology.** Arrows (1) though (4) indicate communication paths between components.

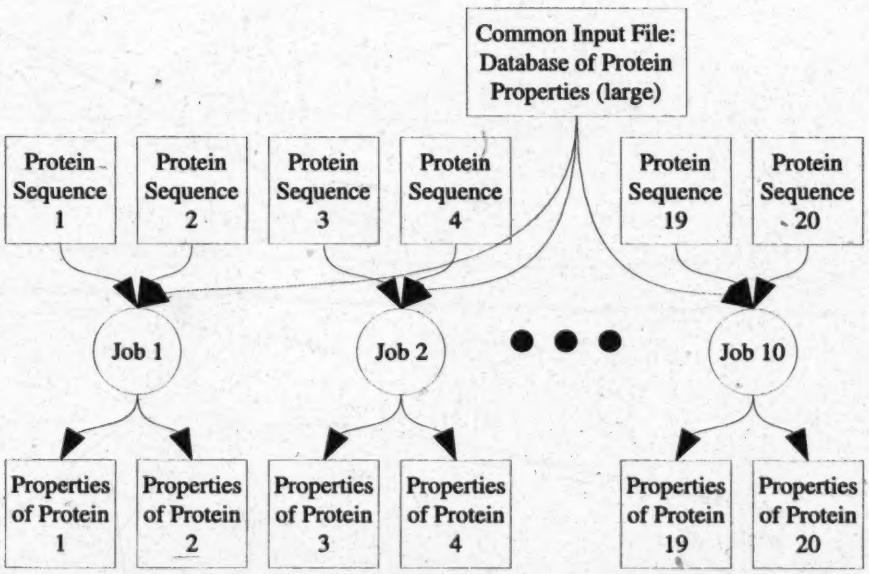


Figure 2.2: **Hypothetical Proteome Analyst Workload Continued: Jobs Access Input and Output Data via the File System.** Rectangles represent files, and circles represent jobs. Here, we show the workload from Figures 1.1 and 1.2 in terms of file access. The individual input and output files are very small, but the common input file (database of protein properties) is large. Proteome Analyst can be used to determine many different properties of proteins. Some of these properties can be computed in parallel by submitting workloads to Trellis, which typically follow the pattern shown here. Two concrete workloads are shown later in Figure 5.2.

long-tailed workload can arise because of failures (e.g., Figure 1.1a), slowly executing placeholders (e.g., Figure 1.2a), or poor load balancing, among other causes.

Long-tailed workloads are bad for makespan. In general, if there are idle resources at the end of a computation, is it often possible to leverage them to help the workload complete sooner. In the case of poor load balancing, this could mean spending more effort to balance or re-balance work. In the case of a failed node or a pathologically slow job execution, redundantly executing the job on an idle placeholder may provide a large benefit.

The long-tail effect is not only important at the end of a workload, but also before barriers [5] or jobs that many other jobs depend on. For example, imagine a workload consisting of 1000 independent jobs, followed by a barrier, and then another 1000 independent jobs. If a pathologically slow placeholder leaves the other placeholders waiting idle at the barrier for an hour, this is just as serious as the same delay at the end of the workload.

2.1.4 Faults and Redundancy

From the perspective of the Trellis scheduler, there are two failure cases to consider for which redundant execution can help with the long tail problem. Note that these two cases do not map exactly to the two motivating examples from Chapter 1; both of those examples correspond to the first case shown below. Addressing many different kinds of failures and performance issues in a consistent way makes the system more robust; recovery does not depend on determining the specific underlying problems.

1. **A placeholder requests a job from the scheduler, and then does not send a "job completed" or "job failed" message to the scheduler for a long or indefinite time.** This can occur because the placeholder node failed, the network between the placeholder and the scheduler failed, or the job is taking pathologically long to complete, among other causes.

   Redundant execution handles all of these cases in the same way. Eventually, the outstanding job is redundantly assigned to another placeholder, and the first placeholder to complete the job is notified that it may persist the job's output. After the persistence step is done, the placeholder notifies the scheduler and the job is considered complete. Failure during the persistence step is detected promptly via a lease mechanism (i.e., the placeholder periodically pings the scheduler and if communication is lost, the placeholder aborts and the scheduler moves the job back to the ready state).

2. **A placeholder requests a job from the scheduler, and then returns a "job failed" message within a reasonable time.** This can occur because the job was specified incorrectly, the placeholder does not have access to necessary programs or data, or there was a network failure between the placeholder and the relevant home nodes, among other causes.

Without redundant execution, failed jobs are retried a user-specified number of times. Redundant execution extends this naturally: If one redundant execution experiences a job failure, the job's retry count is reduced and the other executions continue uninterrupted.

While the redundant execution approach can work around most types of failures in placeholder nodes and the networks that connect them to the rest of the metacomputer, we assume that placeholder nodes do not act maliciously or fail in such a way as to return incorrect or corrupt output. Also, the scheduler and home nodes are single points of failure in the system – forward progress can not be made while they are unavailable.

2.2 Related Work

We present related work in two parts. First, we highlight key similarities and differences with four diverse distributed systems designed to execute computational workloads: Charlotte [2], Satin [33], BAD-FS [3], and MapReduce [7]. Second, we discuss existing file systems with transactional interfaces.

2.2.1 Charlotte

Like our system, Charlotte [2] uses redundant execution to mask the effects of slow or failed nodes participating in a computation. Both systems schedule work using a pull model (i.e., nodes request work; the scheduler does not unilaterally assign it), with web browsers in Charlotte playing an analogous role to Trellis' placeholders. Both systems use a write-back strategy for modified data; computational nodes cache data locally and propagate updates for exactly one successful invocation of each job. Our system distinguishes itself from Charlotte by supporting batch workloads, using a distributed file system for communication between jobs in lieu of shared memory.

2.2.2 Satin

Like Trellis and Charlotte, Satin [33] schedules work with a pull model. However, it is distributed: nodes pull work from other nodes. Satin addresses the issues of slow or failed nodes without using redundant execution. Failed nodes are detected in the communications layer, and slow nodes are handled by subdividing and reassigning work until only a trivially small computation is left for the slow node. Satin is able to subdivide work in this way because it is designed specifically for divide-and-conquer problems, where most of a computation's work is finely split between the leaves of a computation tree. In contrast, our system is not restricted to workloads which can be decomposed into small pieces, and is able to work around faults without explicitly detecting them.

2.2.3 BAD-FS

BAD-FS [3] is an integrated file system and scheduler which helps execute IO-intensive workloads on clusters connected by a wide area network. It leverages unprivileged user accounts and existing batch scheduling systems on these clusters. In these respects, it is very similar to Trellis and the extensions described in this work; we also integrate file system and scheduler to efficiently execute workloads across hosts with ordinary user accounts or clusters with existing batch schedulers. In both systems, jobs access their input and output exclusively via the file system. While Trellis and BAD-FS can both include unmodified applications in their workloads, our system achieves this via bridging file servers (see Figure 2.1 and Section 2.1.1), and thus can not do so without some elevated privilege. BAD-FS avoids this restriction by using an interposition agent to intercept system calls used by jobs to access the file system. Both systems gracefully handle the dynamic addition or removal of resources (hosts, clusters) from the metacomputer.

The most significant difference between BAD-FS and Trellis is that BAD-FS requires detailed dataflow information for a workload in order to execute it. With the exception of the ultimate output files, which must be specified individually, shared input and intermediate data is specified with a volume abstraction. Volumes have a type and size, and are associated with the jobs that access them. BAD-FS leverages this information to (a) provide fault tolerance, (b) minimize the WAN traffic associated with shared input and intermediate data, and (c) manage disk capacity on cluster nodes, which may become scarce as the system tries to avoid communicating over the WAN.

Both systems leverage workflow knowledge to ease file-system consistency. In our work, if the scheduler knows that two jobs are executing concurrently, it assumes they do not make conflicting file-system accesses. Therefore, metadata and data observed by a job may be cached for the duration of that job. BAD-FS makes a similar assumption, but instead of caching, it schedules the concurrent job onto a node in the same cluster where it can access the same storage volume over the cluster LAN. Also, BAD-FS caches input data for the duration of the entire workload, provided there is enough space.

To achieve fault tolerance, BAD-FS re-executes jobs when jobs, nodes executing jobs, or storage servers fail. Failed nodes are detected by heartbeat messages [27] from a catalog server. A volume is considered lost if either the node hosting it becomes inaccessible or a job that can write to the volume fails. In this way, failure in one job can trigger re-execution of previous jobs. To mitigate this effect, BAD-FS replicates volumes between jobs when it is statistically cheaper to copy data than re-execute completed jobs after a future failure. Trellis does not re-execute complete jobs, as their output is already stored on the home nodes, which Trellis does not protect. Similarly, once BAD-FS has extracted output data for a pipeline and copied it to the home file system, the data is no longer protected against loss by re-execution. In contrast to our work, BAD-FS does not protect against slow job executions, and therefore these executions can needlessly prolong makespan (the long tail effect, see Section 2.1.3). For example, this contributes to the reduced job completion rate

in the last hour of workload execution in Figure 9 of the BAD-FS paper [3].

BAD-FS takes advantage of dataflow information to reduce WAN traffic. Input data is cached cooperatively by nodes in each cluster, and intermediate data is never transferred over the WAN. The scheduler works with the file system to ensure potentially limited cluster storage is used efficiently. In this work, we manually copy shared input data to placeholder nodes before computation begins, and do not optimize for locality of intermediate data (but this was explored elsewhere [12]). Trellis uses per-placeholder, whole-file caching for all data. BAD-FS uses per-cluster, block-level caching for read-only input data.

2.2.4 MapReduce

MapReduce [7] is a programming model and system which allows an interesting class of problems to be specified simply and efficiently executed on a cluster. MapReduce scales very well; one computation can process terabytes of data on a cluster with thousands of nodes. It is expressive and efficient enough to be used for many computational tasks at Google, including maintaining their production web index.

A MapReduce program consists primarily of two functions. The *map* function reads input key/value pairs and maps them to intermediate key/value pairs. The *reduce* function reads the values associated with each intermediate key, and produces output key/value pairs. The map operation is 1:n (i.e., zero or more key/value pairs are produced for every input key/value pair), and the reduce operation is n:m (i.e., the set of values associated with each intermediate key is replaced with a new set of values, typically of size zero or one). As with Satin, this functional programming model has the nice property that function invocations are independent of each other and can be executed concurrently or repeatedly, without interfering with each other.

In the MapReduce system, input and intermediate data are partitioned and a single master manages the assignment of the associated map and reduce tasks to a set of nodes. Load balancing is simple; each task is reasonably fine grain, and tasks are assigned to nodes as previous tasks complete. To provide fault tolerance, the master periodically pings each worker node, and reassigns affected tasks when nodes fail (a failed node requires the re-execution of that node's in-progress tasks, as well as all of the map tasks the node previously completed). As with our system, MapReduce contains a single point of failure: If the master fails, the entire computation aborts.

As in Charlotte and our work, as a MapReduce computation nears completion, outstanding map and reduce tasks are redundantly executed; one "backup task" is initiated for each outstanding task. A given task is considered complete when either the original task execution or the backup task execution completes. The authors observe that task executions can unexpectedly become extremely slow for a number of reasons, including failing disks, resource contention with other jobs, or faulty configuration. One slow task execution can dramatically delay workload completion. In one case, disabling the "backup task" mechanism increased makespan by 44% [7].

When tasks are redundantly executed, it is important to ensure only one execution persists its output. MapReduce achieves this by writing output atomically, using a different strategy for intermediate (map) output and final (reduce) output. When a map task completes, its output is stored in a set of unique temporary files. The paths to these files are communicated to the master, which updates its data structures atomically before passing the information to reduce tasks. If a redundant execution for the same map task later completes, the master ignores it. Similarly, each reduce task outputs its data to a single temporary file. On completion, this output file is atomically renamed [9] to the final output file path. In our extended version of Trellis, a job's output can not be atomically persisted. This is because we use unmodified file systems to store data on home nodes, and jobs are allowed to make arbitrary changes to these file systems. Most file systems do not provide an interface to perform a set of operations atomically, and we do not assume Trellis has exclusive access to them. To work around this limitation, jobs are executed in two phases. In the first phase, redundant executions of a job cache all file-system modifications locally. In the second phase, one execution is selected by the scheduler to persist its output. Failure during the second phase is detected by a lease mechanism, and results in the job being re-executed from scratch.

One weakness of Charlotte's eager-scheduling strategy, which both MapReduce and our system build upon, is that if some task consistently fails (e.g., due to a deterministic bug), the workload will never complete [7]. MapReduce overcomes this issue by suppressing records that repeatedly lead to program crashes. Trellis simply gives up executing a job after a user-specified number of retries.

2.2.5 Transactional File Systems

In this work, we use transaction-style semantics to control file-system caches associated with each placeholder. As a job runs, reads and writes of file-system metadata and data are cached locally. A *rollback* command clears the local cache, and a *commit* command persists pending modifications to home nodes. This allows for isolation between redundant executions of the same job, which will likely access the same file-system paths. However, full ACID [26] semantics (i.e., atomicity, consistency, isolation, and durability) are not provided. In particular, updates are not atomic, and concurrent executions of different jobs are not isolated – they can see each other's updates in arbitrary order.

A number of existing systems provide transactional interfaces to file systems including QuickSilver [24], Inversion [17], and Amino [32]. Inversion and Amino can be configured to provide full ACID semantics. QuickSilver extends the transaction concept across many services in a distributed operating system, but does not use this capability for redundant execution of jobs.

2.3 Concluding Remarks

Components of a distributed system can fail or suddenly become slow at any time. We have shown how a variety of systems attempt to mitigate the resulting performance and availability challenges. In

this work, we combine redundant execution with a transaction-style file-system interface to mitigate the effect of failed or slow nodes while executing batch workloads.

Chapter 3

Architecture

In the previous chapter, we explored performance challenges faced by metacomputing systems when participating nodes fail or become slow. We then described how a variety of systems address these challenges. In particular, we introduced the long tail problem, and showed how redundant execution works well to overcome it and compares favorably to other approaches. In this chapter, we present the new system architecture and the design of the new features.

3.1 Three Deployment Strategies for PA/Trellis

We present the Trellis architecture in the context of Proteome Analyst (PA); the combined system is called *PA/Trellis*. This allows us to discuss the architecture in terms of a concrete workload, and sets the stage for the evaluation in Chapter 5. Focusing on alternatives for providing distributed file-system services, three variations of the architecture are presented: one using NFS (Figure 3.1), one using the Trellis File System library (Figure 3.2), and one using the Trellis Samba (Figure 3.3). The first and third approaches work with unmodified applications, and the last two approaches support redundant execution and work across administrative domains. Observe that Trellis is effectively two distributed systems working in tandem: a scheduling system and a distributed file system. These systems are loosely coupled to provide redundant execution. In the following sections, we provide a brief overview of the components from Figures 3.1, 3.2, and 3.3.

3.1.1 Proteome Analyst

Proteome Analyst [29] is a web-based application which annotates user-supplied proteomes with biological properties. A *proteome* is the set of proteins encoded by an organism's DNA. Proteins in the set are expressed as sequences of amino acids, each of which are denoted by letters of the alphabet. PA uses various databases and tools to gather information about each protein, and then leverages this information to make higher-level classifications, including subcellular location and function.

The PA web application is written in Java and is hosted by a Tomcat application server (Fig-

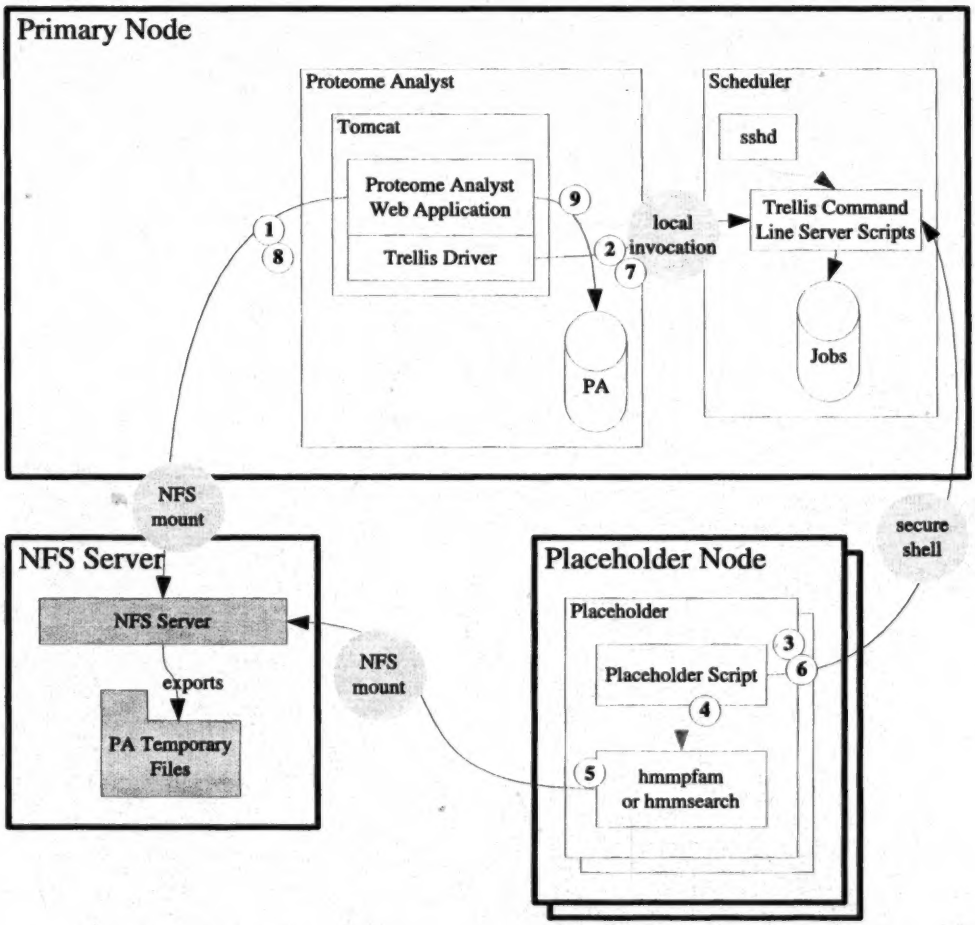


Figure 3.1: **Architecture of PA/Trellis with NFS.** Communication paths between subsystems are annotated with light gray circles. File-system components are shaded dark gray. (1) Proteome Analyst writes input files to a shared directory on the NFS server. (2) Proteome Analyst, via Trellis Driver, submits a set of jobs for the placeholders to execute. (3) Placeholder scripts poll the scheduler for work. (4) A placeholder invokes `hmmpfam` or `hmmsearch` as per a job description (command line). (5) This program accesses input and output files via the shared directory on the NFS server. (6) The placeholder notifies the scheduler that the job is complete. (7) The job submission script completes, returning control to a Trellis Driver thread. (8) Once all of the jobs have completed, PA reads the output files from the shared directory, transforming them to hmmpfam-style output if `hmmsearch` was used. (9) PA writes the final output to its database.

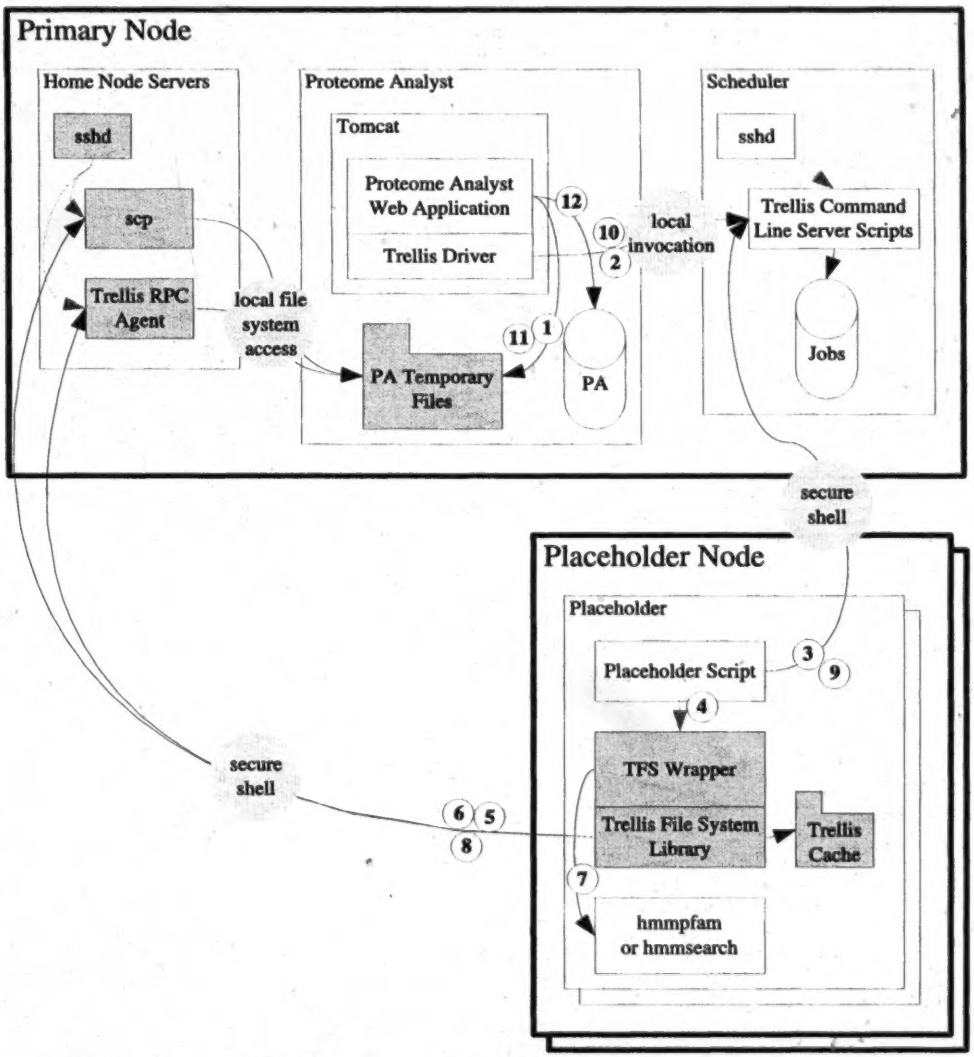


Figure 3.2: **Architecture of PA/Trellis with the Trellis File System Library.** Communication paths between subsystems are annotated with light gray circles. File-system components are shaded dark gray. (1) Proteome Analyst writes input files to a local directory. (2) Proteome Analyst, via Trellis Driver, submits a set of jobs for the placeholders to execute. (3) Placeholder scripts poll the scheduler for work. (4) A placeholder invokes the TFS wrapper program as per a job description (command line). (5) The TFS wrapper determines the input and output file paths from its command-line arguments. To query metadata about these paths, the embedded Trellis File System library starts an instance of the TFS RPC agent on the primary node, and sends it requests via the `ssh` channel. (6) The input file is copied, via `scp`, from the primary node to the Trellis cache. (7) The TFS wrapper invokes `hmmpfam` or `hmmsearch`, as per its arguments, redirecting standard input from the input file in the Trellis cache, and standard output to a new file in the Trellis cache. (8) The output file is copied, via `scp`, to the primary node. (9) The placeholder notifies the scheduler that the job is complete. (10) The job submission script completes, returning control to a Trellis Driver thread. (11) Once all of the jobs have completed, PA reads the output files from the local directory, transforming them to hmmpfam-style output if `hmmsearch` was used. (12) PA writes the final output to its database.

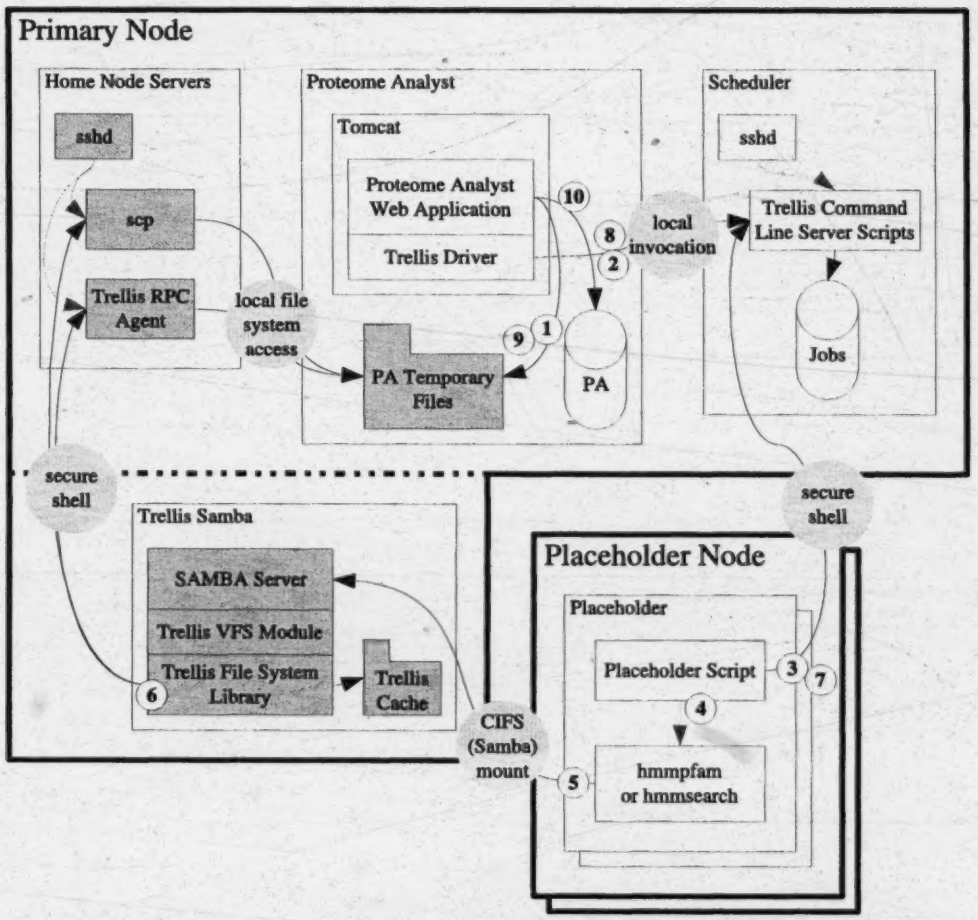


Figure 3.3: **Architecture of PA/Trellis with Trellis Samba.** Communication paths between subsystems are annotated with light gray circles. File-system components are shaded dark gray. (1) Proteome Analyst writes input files to a local directory. (2) Proteome Analyst, via Trellis Driver, submits a set of jobs for the placeholders to execute. (3) Placeholder scripts poll the scheduler for work. (4) A placeholder invokes `hmmpfam` or `hmmsearch` as per a job description (command line). (5) This program accesses input and output files via the Samba server. (6) The Samba server accesses the "PA Temporary Files" directory on the primary node by sending requests to the Trellis RPC Agent and by copying files with `scp`. The Trellis RPC Agent is started once, via `ssh`, on the first request. (7) The placeholder notifies the scheduler that the job is complete. (8) The job submission script completes, returning control to a Trellis Driver thread. (9) Once all of the jobs have completed, PA reads the output files from the local directory, transforming them to hmmpfam-style output if `hmmsearch` was used. (10) PA writes the final output to its database.

ure 3.1, top center). However, it spends most of its computation time in command-line tools. Often, input and output data are passed between the web application and the command-line tools via files. This makes PA a nice fit for Trellis, and it was previously extended to use Trellis to execute the BLAST [1] and HMMER [11] components of its workload, using NFS to access input and output data over a local network [12, 4]. In this work, we focus on the HMMER tool, extending support to allow it to work with the Trellis File System (directly, or indirectly via Samba) in addition to NFS.

PA uses the Trellis Driver [13] package to submit jobs to a Trellis metacomputer and await their completion. When a user submits a proteome, PA creates a set of input files, submits a set of HMMER jobs to Trellis, waits for all of these jobs to complete, reads the corresponding set of output files, updates its database, and continues with its other computations (Figure 3.1, Steps 1, 2, 7, 8, and 9, respectively).

3.1.2 Trellis Driver

Trellis Driver [13] is a Java package which allows arbitrary Java programs to submit jobs to a Trellis metacomputer, and then wait for individual jobs or groups of jobs to complete. Submitting work (Figure 3.4) is easy and follows the same paradigm as Java's `Runtime.exec()` method. Trellis Driver communicates with the Trellis scheduler by executing the `mqsub` job submission script in blocking mode and waiting for its completion (Figure 3.1, Steps 2 and 7). In this mode, `mqsub` does not return until the submitted job is complete, which it determines by polling. As `mqsub` is executed locally, applications using Trellis Driver must be on the same host as the Trellis scheduler. The maximum number of outstanding jobs is configurable, and one thread is created for each. Most of the time, these threads are waiting for a monitor or are blocked on `mqsub`. Trellis Driver can be configured to batch jobs together before submitting them, which helps amortize overheads when the jobs are fine grain.

3.1.3 Placeholder Scheduling System

With placeholder scheduling [19, 18], users submit jobs along with dependencies to a central scheduler, and may subsequently query for job status. Placeholders periodically pull work from the scheduler and notify it when the work is done. In PA/Trellis, Trellis Driver plays the role of user, submitting jobs and querying for their status.

The scheduler consists of a job database and a set of scripts (Figure 3.1, top right), which users and placeholders invoke to interact with the scheduler. Placeholders consist of scripts that sequentially query for an available job, execute that job, and report the job's exit code (Figure 3.1, Steps 3, 4, and 6, respectively).

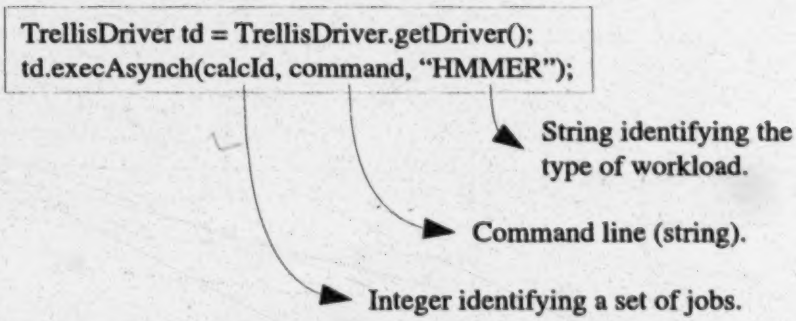

TrellisDriver td = TrellisDriver.getDriver();
td.execAsynch(calcId, command, "HMMER");
String identifying the type of workload.
Command line (string).
Integer identifying a set of jobs.

Figure 3.4: **Code Fragment: Submitting Work to a Trellis Metacomputer with Trellis Driver.** Java programs (here, PA) use Trellis Driver to submit work to a Trellis metacomputer in the form of command lines. In this example, `calcId` identifies a workload so that a subsequent barrier waits only for jobs from the appropriate workload, and the string "HMMER" allows parameters, such as batching factor, to be configured differently for different types of workloads.

3.1.4 HMMER

Profile hidden Markov models (HMMs) can be used to capture common patterns from families of proteins that share a common property. If a new protein matches the HMM, then it may also share the property (i.e., belong to the protein family the HMM was created from). It is useful to aggregate these HMMs into large databases of protein families (e.g., Pfam [8]), so that unknown proteins can be classified by comparison with a database. HMMER [11] is a suite of tools that can do exactly this comparison. PA uses HMMER with Pfam to classify unknown proteins by protein family. Specifically, it uses the `hmmpfam` tool to compare individual proteins with all HMMs in Pfam, or the `hmmsearch` tool to compare individual HMMs with all proteins in a given proteome [4].

In PA/Trellis, HMMER accesses its input and output data via a file system. The database is static and large, and is therefore manually copied to each machine and accessed locally. The other input and output files may be accessed via NFS or the Trellis File System (via a wrapper program linked to the TFS library or a suitably extended Samba server).

3.1.5 NFS

For the NFS configuration (Figure 3.1) we use the standard Linux NFS server. The NFS server exports local directories, which are mounted by all participating nodes (the primary node with PA and the Trellis scheduler, as well as all of the placeholder nodes). All clients can transparently use the same paths to access input and output files, which are stored on the NFS server.

3.1.6 Trellis File System

The Trellis File System [25] is a user-level, overlay distributed file system, targeted at executing high-performance-computing (HPC) workloads efficiently. The core component of TFS is the TFS

library. This library provides a file-system API similar to POSIX; applications written or modified to use this API may be linked with the TFS library to access the file system. In Figure 3.2, a wrapper program allows the unmodified HMMER tools to access data via TFS. In Figure 3.3, the unmodified HMMER tools access data via a Samba server extended to mediate access to the TFS library (see Section 3.1.7).

In TFS, files are named by *secure copy locators (SCLs)*. Each SCL includes a protocol, username, host, and local path, essentially naming a path on a remote file system. For example, the SCL `scp:theuser@thehost:/directory/file` names the file `directory/file` on host `thehost` when accessed by the user `theuser` via `ssh`. The remote host is called a *home node*. TFS uses whole-file caching; the cache is labeled "Trellis Cache" in Figures 3.2 and 3.3. TFS currently includes support for three home-node protocols: secure shell with SFTP, secure shell with Trellis RPC (a custom protocol), and CIFS.

3.1.7 Trellis Samba

Samba [23] is an open-source CIFS file server. Typically, the CIFS protocol is used for file sharing between Windows clients and servers, but support is available on Linux and other systems as well. Samba has a Virtual File System (VFS) layer which allows custom file systems to be exported. A new Samba VFS module (Trellis VFS Module, Figure 3.3) was created to provide indirect access to the Trellis File System. Standard CIFS clients (here, the smbfs Linux kernel module) can mount a Samba share allowing unmodified applications access to TFS.

The extended Samba server, Trellis Samba, is an example of a bridging file server [14]. LAN protocols (i.e., CIFS) are used between the client and the file server, and WAN protocols (e.g., secure shell) are used between the file server and the home node.

3.2 A Framework for Multiple Write-back Policies in the Trellis File System Library

The TFS library has been extended to support multiple write-back policies. The *implicit write-back* policy provides traditional TFS semantics: metadata changes are propagated to home nodes as they are made and data changes are propagated when files are closed. The *explicit write-back* policy provides support for the redundant execution feature; all changes are cached locally and propagation is explicitly controlled. The scheduling system leverages the explicit write-back policy to isolate redundant executions of the same job from each other.

3.2.1 Deviation from POSIX Semantics

TFS does not support the complete POSIX API or all of its semantics. Other commonly used distributed file systems (e.g., NFS) deviate from POSIX as well. Some limitations result from the fact that TFS is an overlay distributed file system, using other file systems as its backing store. Other

limitations result from our focus on HPC workloads; our primary goal is to execute these workloads efficiently. The following are three key deviations from the POSIX standard:

1. For performance and simplicity, TFS uses whole-file caching, and performs all operations on open files and directories in the local cache. Updated versions are propagated back to home nodes when files are closed. This results in *close-to-open consistency*. One consequence is that if two concurrent writers update the same file, the second writer to close the file will overwrite the changes made by the first. A related consequence is that external changes to a file are not visible while that file is open.

2. TFS addresses files and directories on home nodes with paths, not inodes. Therefore, it can not detect certain kinds of external changes, including moves or deletions of open files, and may function incorrectly in these cases. Also, TFS may communicate with a home node multiple times in order to satisfy one request (e.g., determining that a path is a file and then copying it to the cache in response to an open request). Together, these design decisions lead to intrinsic race conditions.

3. Only regular files and directories are supported. Symbolic and hard links are not. Symbolic links on home nodes appear as regular files and directories. This has the consequence that two paths may point to the same underlying file, and these paths will be cached and updated independently. Symbolic link support is planned for a future version of TFS.

3.2.2 Interaction with the New Cache Manager

The TFS library is composed of three components:

1. **TFS Layer.** The TFS layer provides the client-facing file-system API. This API is modeled after the POSIX file-system API and consists of functions with similar names, arguments, and semantics. Most operations are performed indirectly on the Trellis cache, which mirrors portions of home-node file systems. TFS notifies the cache manager before it reads any metadata or data from the cache (so that the cache manager has a chance to bring it up-to-date) and after it modifies any metadata or data in the cache (so that the cache manager has a chance to propagate the changes to home nodes).

2. **Cache Manager.** The cache manager keeps track of files and directories in the cache, and decides when to copy metadata and data to or from home nodes. Communication with home nodes is handled through the RPC (remote procedure call) layer. By centralizing control of the cache in one place, it becomes possible to introduce multiple cache policies.

3. **RPC Layer.** The RPC layer provides an abstract interface to query for metadata on home nodes, copy files between the local cache and home nodes, and to perform remote metadata operations. It encapsulates support for a variety of home-node protocols.

	When metadata is changed e.g., mkdir()	When data is changed e.g., write()	When a file is closed i.e., close()	When a commit is requested
Implicit Write-back	✓		✓	N/A
Explicit Write-back				✓

Table 3.1: **Write-back Policies.** Check marks indicate when associated changes are propagated to home nodes. Implicit write-back propagates metadata changes immediately and data changes when associated files are closed. Explicit write-back propagates all changes on commit.

When client code invokes a function in the TFS layer, the TFS layer does the following:

1. **Update Metadata.** The TFS layer asks the cache manager to ensure that the relevant cache paths have up-to-date metadata. The cache manager returns the state of these paths, which indicates their type, whether the corresponding data is also current, and whether the paths are open.

2. **Validate Metadata.** The TFS layer validates that the paths are in legal states. For example, for the `rename(a,b)` operation, `b` must be non-existent or a closed file.

3. **Update data.** If necessary, the TFS layer asks the cache manager to bring the paths' data up to date. For example, `rmdir()` requires the directory to be synchronized in order to verify that it is empty, and `open()` requires that the contents of an existing file are current if the `O_TRUNC` flag is not specified.

4. **Perform operation in cache.** The TFS layer performs the requested operation (e.g., `stat()`, `open()`, `close()`, `unlink()`, `rename()`, etc.) on the cache path(s).

5. **Signal changes.** If the operation results in a change to the file system, the TFS layer notifies the cache manager so that the change may be reflected on the corresponding home node, as appropriate.

3.2.3 Write-back Policies

The following write-back policies are summarized in Table 3.1:

Implicit Write-back

The implicit write-back policy is intended for situations where the home node must be updated before a file-system operation can be considered complete, and concurrent changes to home-node file systems must be visible. This policy is similar to the classic TFS behavior and is a conservative default.

With the exception of operations on open files or directories, home nodes are accessed for every file-system operation. An optional metadata cache allows metadata reads to be cached locally for a configurable amount of time.

Explicit Write-back

The explicit write-back policy is intended for situations where a series of file-system reads and writes are guaranteed not to conflict with any other client, and it is desirable to explicitly control when changes are persisted to home nodes. Reads of metadata and data are indefinitely cached, avoiding redundant requests. Writes of metadata and data are also cached, and are not immediately propagated to home nodes. After a series of file-system changes have been made, the changes may be discarded (rollback) or persisted together (commit).

The explicit write-back policy works well in the context of batch scheduling because the user is already required to tell the scheduler about dependencies between jobs. For example, imagine that there are three jobs – A, B, and C – and Job A writes a file which Job B subsequently reads. In this case, the user specifies that Job A must run before Job B. By implication, Jobs A and C may run concurrently, and do not access the file system in conflicting ways.

In spite of dependency information, it may happen that one job will observe file-system paths modified by a concurrently executing job. For example, one job might list the contents of a directory that a concurrently running job creates a file in, even if the first job does not care about the presence of that file. In these cases, due to caching, concurrently running jobs may see only a subset of each other's updates. If the dependency information has been incorrectly specified, the resulting race conditions may lead to failures or incorrect output, but this is true even without explicit write-back.

3.3 Scheduling for Isolated, Redundant Execution

The Trellis scheduler has been extended to support redundant execution. Jobs are redundantly executed whenever placeholders would otherwise go idle. The scheduler leverages TFS' explicit write-back policy to ensure that concurrent executions of the same job do not observe each other's file-system changes, and that only one of these executions persists its output.

As jobs proceed through the scheduling process, they normally transition through the following states: waiting, ready, running, and done. To enable redundant execution, a new *exiting* state has been created between running and done. A job in the running state may be executing redundantly on multiple placeholders. After the first of these placeholders completes, the job moves into the exiting state, and the corresponding file-system changes are propagated to the home nodes. Other placeholders executing the same job continue uninterrupted, but their output is not used. After propagation is complete, the job moves into the done state. Jobs executed without isolation are not redundantly executed and follow the original state transitions. Figure 3.5 illustrates the new states and transitions.

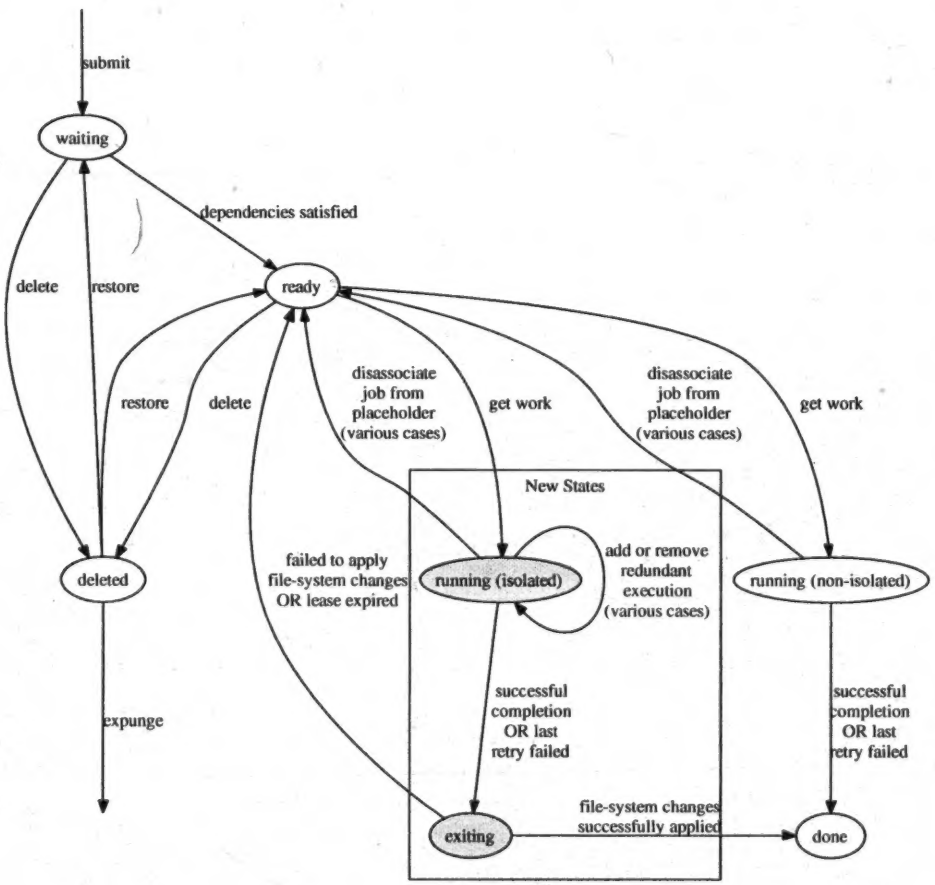


Figure 3.5: **Job State Transitions.** The two new states are shaded and boxed. The other states and transitions have not changed.

3.3.1 Job Selection Policy

The Trellis scheduler works on a pull model. Periodically, placeholders request work (command lines) to execute. At these times, the scheduler may select a ready job for execution, select a running job for redundant execution, ask the placeholder to come back later, or ask the placeholder to exit. Thus, there is an opportunity to evaluate different policies for job selection.

We use a simple policy to control job selection. Ready jobs are selected first, with preference given to jobs with higher priority or earlier submission time. Once there are no more ready jobs to chose from, the running jobs are redundantly executed in the same preferential order. This pattern continues to an arbitrary level of redundancy: if all jobs are executing redundantly on two placeholders, and another placeholder asks for work, its assigned job will be running with triple redundancy.

Redundantly executing a job has the potential to reduce the time for that job to complete, if the new execution has a chance of completing before the original one. This is the case if the original placeholder has silently failed (e.g., due to network failure) or if the original placeholder is executing the job slowly relative to the new placeholder. If redundant execution is able to reduce the completion time for a job, it may lead to reduced makespan. This is the case if the workload is almost complete (or similarly, if there are few jobs remaining before a barrier), or if the selected job is on the critical path. Therefore, redundant execution is attractive for jobs which are taking pathologically long to complete (perhaps due to placeholder failure) near the end of the workload.

Opposing this potential benefit is a decrease in efficiency. Redundantly executed jobs consume computing resources that could be used for something else: other Trellis jobs or work external to the metacomputer. The current implementation does not include a way to abort redundant executions when they are no longer useful (i.e., when another execution finishes before them), which adds to the significance of this issue.

Nonetheless, the overall opportunity can be large. In one example with MapReduce, makespan was 44% longer without redundant execution, with redundant execution tuned to increase the total resource consumption by only a few percent [7].

3.3.2 Scheduler and File System Integration

Placeholders drive the interaction between the file system and scheduler. To communicate with the scheduler, placeholders use `ssh` to invoke scripts remotely. To communicate with the Trellis File System, placeholders open special files (in the Samba case, as in Figure 3.3) or locally invoke TFS utilities (in the TFS case, as in Figure 3.2). For each job execution, a placeholder follows the following steps:

1. **Request a job from the scheduler.** The job is now in the running state.
2. **Run the job.** As the job accesses the file system, reads and writes of metadata and data are cached.

3. **Notify the scheduler that the job is complete.** The scheduler decides whether the associated output should be persisted or not.

4. **If the scheduler says no, tell TFS to abort.** The job is now in the ready state. Pending file-system modifications are discarded, and cached metadata and data are marked potentially invalid. The home-node file systems remain unchanged. Execution continues from Step 1.

5. **Or, if the scheduler says yes, tell TFS to commit.** The job is now in the exiting state. TFS propagates the file-system modifications to the home nodes, moving data as required. At this point, failures are not protected by redundant execution. As an alternative, the placeholder passes information to TFS so that it can coordinate a lease with the scheduler.

6. **Tell the scheduler whether or not the output was persisted successfully.** The job is now in the done state (for success) or the ready state (for failure).

In the CISS [20] experiments, occasionally failures to transfer data would result in only a subset of a job's output files being persisted on home nodes. Without redundant execution and the explicit write-back policy, these failures are reported to user programs as failures in the `close()` system call, which are often ignored. One of the advantages of our approach is that failure to copy output data or make changes on home nodes is detected during the commit step and reported to the placeholder, which then notifies the scheduler (Step 6, above). This reduces the chance that a job will be considered complete while its output is only partially persisted. This difference arises because with explicit write-back, placeholders are responsible for propagating file-system modifications to home nodes, and can therefore detect associated failures and abort job execution.

3.4 Concluding Remarks

In this chapter, we have presented our extensions to the Trellis File System and scheduler to support redundant execution, in the context of PA/Trellis. Redundant execution requires isolation between executions of the same job, which we achieve through a lightweight integration of file system and scheduler. The PA/Trellis configurations presented here (Figures 3.1, 3.2, and 3.3) are reused in Chapter 5 for quantitative evaluation.

For some workloads, or with large metacomputers, our simple job selection policy may be overly aggressive. The following are some alternate policies that could be explored as part of future work:

1. **Bound the degree of redundancy.** For example, do not execute any job with more than double redundancy, or limit the total number of redundant executions that may be ongoing at any one time.

2. **Only redundantly execute jobs in the tail of the workload.** The tail region could be defined based on the number of jobs remaining in the workload or before a barrier. The simple policy

behaves this way when the number of jobs in a workload is much larger than the available degree of concurrency, which is common.

As an extension to this strategy, file-system isolation could be disabled in the non-tail region of the workload, which would eliminate overheads caused by the extra transition to the exiting state, as well as any overheads caused by the explicit write-back policy. However, redundant execution could not be used to recover from failures in this case, and some other mechanism would be required.

3. **Use a heartbeat to detect failed or unreachable placeholders.** In this case, the placeholder would drive the heartbeat, connecting the scheduler, as the reverse connection might not be possible (see Figure 2.1). If the heartbeat mechanism detected a failure, the placeholder could abort its job, and the scheduler could reassign it.

Chapter 4

Implementation

In the previous chapter, we presented the design of our extensions to the Trellis File System and scheduler in the context of PA/Trellis. In this chapter, we present the implementation of these extensions, as well as other modifications to the overall PA/Trellis system. In the next chapter, we leverage this system to quantitatively evaluate the performance impact of redundant execution.

4.1 Extensions to the Trellis File System

In Section 3.2.2, we described how the TFS layer communicates with the cache manager in the TFS library at a high level. In Table 4.1, we list the calls that comprise the cache manager interface. Implementations of this interface are provided for the implicit and explicit write-back policies.

The two calls specific to the explicit write-back policy, to rollback or commit pending changes, are exposed to placeholders via a command-line utility (in the case of applications linked directly to the TFS library), and via special files (in the case of applications accessing data via Trellis Samba). The command-line utility can be configured to periodically renew a lease (by calling an arbitrary command) while performing a commit, aborting should the lease expire.

4.2 Extensions to the Placeholder Scheduling System

In this section, we discuss changes to the Trellis scheduler and placeholders to support redundant execution of jobs.

4.2.1 Database Schema

The scheduler maintains all of its state in a database, which we extend slightly to support redundant execution. In the database schema (Figure 4.1), there is one table for each job state, among other tables. Three changes were made to the schema:

1. A boolean, `isolated`, was added to the `running` table. Each job in the running state may be executing with or without file-system isolation, corresponding to the "running (isolated)"

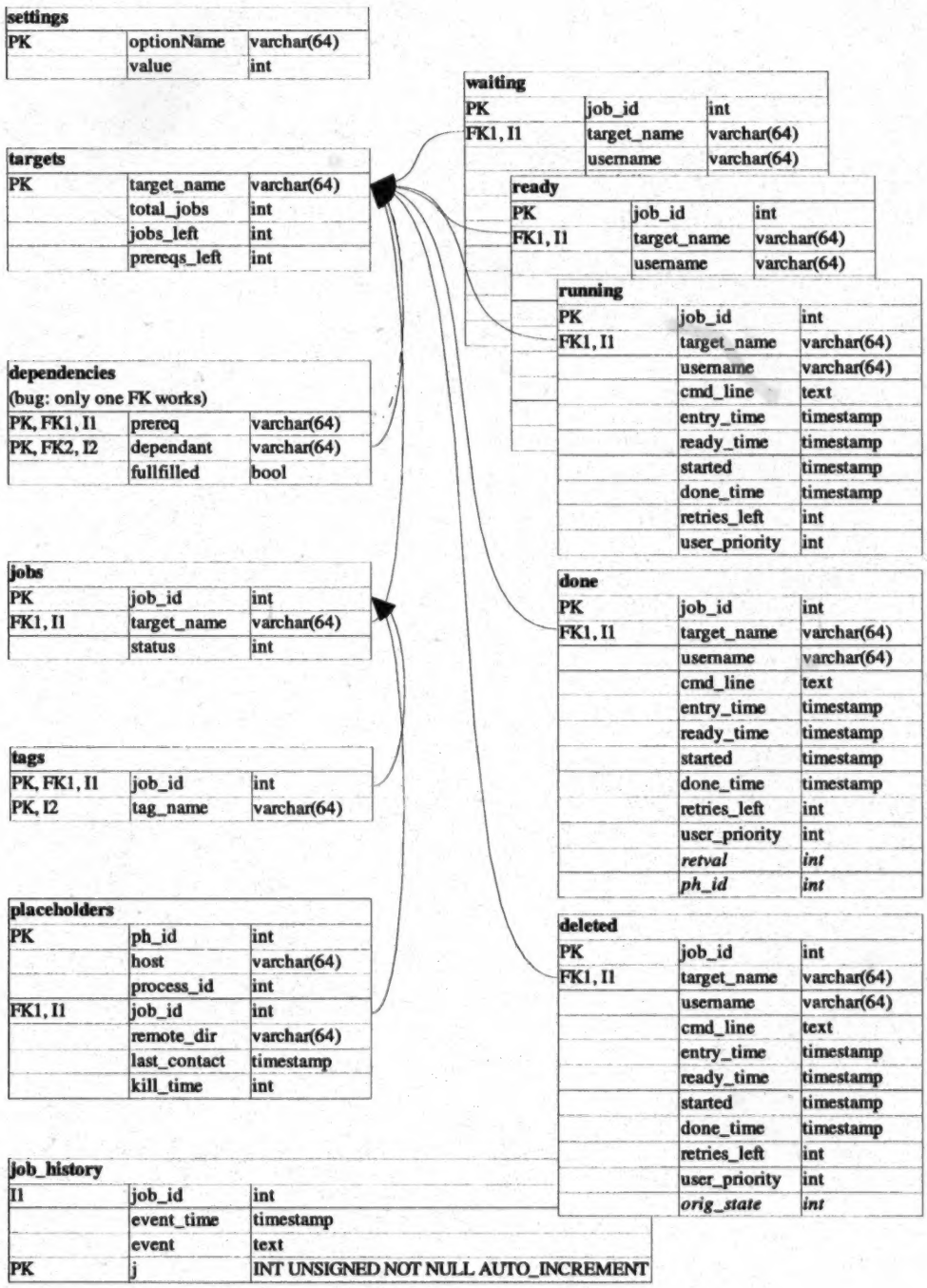


Figure 4.1: Scheduler Database Schema (original)

Call	Purpose
sync_meta()	Ensure the metadata for a cache path is up-to-date.
sync_file()	Ensure the contents of a file in the cache are up-to-date.
cache_file_opened()	Notify the cache manager that a file has been opened in the cache.
cache_file_closed()	Notify the cache manager that a cache file has been closed.
sync_directory()	Ensure the metadata for a directory and all of its immediate children are up-to-date.
cache_directory_opened()	Notify the cache manager that a directory has been opened in the cache.
cache_directory_closed()	Notify the cache manager that a cache directory has been closed.
cache_mkdir()	Notify the cache manager that a directory has been created in the cache.
cache_rmdir()	Notify the cache manager that a directory has been deleted in the cache.
cache_unlink()	Notify the cache manager that a file has been deleted in the cache.
cache_rename()	Notify the cache manager that a path has been moved or renamed in the cache.
tcmgr_de_reset()	**Rollback.** For the explicit write-back policy only, discard pending file-system modifications and mark all cache paths as potentially invalid.
tcmgr_de_commit()	**Commit.** For the explicit write-back policy only, persist pending file-system modifications to home nodes, moving data as required.

Table 4.1: **Cache Manager Interface**

and "running (non-isolated)" states in Figure 3.5, and this flag distinguishes the two cases. Only jobs executing with file-system isolation may be redundantly executed.

2. A new table, `exiting`, was added with the same fields as the existing `done` table. This corresponds to the new "exiting" state in Figure 3.5. The extra fields, `retval` and `ph_id`, previously unique to the `done` table, store the job's exit code and the id of the placeholder which completed the work; once file-system modifications are successfully propagated to home nodes, these values are copied to the `done` table.

3. The meaning of the `started` timestamp has changed slightly. In the presence of redundant execution, this timestamp refers to the moment the job most recently entered the running state, which may be different than the moment when the successful execution began.

4.2.2 Placeholder-facing Interface

The scheduler's *placeholder-facing interface* is the set of scripts, along with their input, output, and parameters, invoked by placeholders to communicate with the scheduler. The protocol used to communicate between placeholders and the scheduler has been extended to support the requests outlined in Section 3.3.2 for redundant execution. The zero-infrastructure placeholder uses this extended protocol, and supports redundant execution, while the other placeholder types continue to use the existing protocol, and do not support redundant execution.

Interface Script	getNextJob.py
Purpose	request a work item (command line)
Original	
Arguments	<placeholder id> <server-side Trellis directory> <queue>
Exit Code	0 = job returned; 1 = retry later; 2 = exit
Output	command line, if applicable
Extended	
Arguments	optional fourth argument: 0 = does not support isolated execution (default); 1 = does support isolated execution
Exit Code	no change
Output	If the placeholder does not support isolated execution, the output is the same as above (i.e., the command line when there is available work). If the placeholder does support isolated execution, a 0 or 1 is prepended to this command line to indicate whether or not the job should be executed in isolation (1=yes).

Table 4.2: **Placeholder Interface: Request Job**

Scheduler interface changes are detailed in Tables 4.2 (request job), 4.3 (job done), 4.4 (renew lease), and 4.5 (commit done). Placeholders communicate with the scheduler by connecting to it with secure shell and invoking scripts. Input is provided via command-line arguments, and results are returned via exit codes and standard output. This communication mechanism between placeholders and the scheduler was discussed in Figure 4, Section 4 of our journal article on the Trellis Security Infrastructure [15].

4.2.3 User Interface

Similar to the placeholder-facing interface, the scheduler exposes a user interface via a set of scripts. For example, the `mqsub` command (introduced in Section 3.1.2) submits a job, and the `mqstat` command displays job status. This interface has been extended naturally; jobs in the exiting state are reported as such, and natural operations (e.g., selecting or restarting jobs) have been extended to support the new states.

4.3 TFS Wrapper

As a middle ground between modifying applications to access data via the TFS library, and supporting unmodified applications via Trellis Samba, a new wrapper program was created. The program takes an input SCL (secure copy locator; see Section 3.1.6), an output SCL, and a command line. The command line is run, with the contents of the first SCL piped into standard input, and standard output redirected into the second SCL. In this way, unmodified applications that can access their input and output data exclusively via standard input and standard output can work with the TFS

Interface Script	jobDone.py
Purpose	notify scheduler that work item is done
Original	
Arguments	<placeholder id> <server-side Trellis directory> <queue> <job's exit code>
Exit Code	0 = success; anything else = retry
Output	None
Extended	
Arguments	no change
Exit Code	no change
Output	If the associated job was executed in isolated mode, a boolean (0/1) may be written to indicate whether or not the associated file-system changes should be applied. If it is omitted, file-system changes should not be applied. (This is an asymmetry in the current implementation due to the way redundant executions are marked when they lose the race to complete first.)

Table 4.3: **Placeholder Interface: Job Done**

Interface Script	renewExitingLease.py (new)
Purpose	notify scheduler that the commit step is ongoing
Arguments	<placeholder id> <server-side Trellis directory> <queue>
Exit Code	0 = success; anything else = lease was not renewed
Output	None

Table 4.4: **Placeholder Interface: Renew Lease.** The timing parameters are hard coded: The scheduler considers the lease expired 6 minutes after the last successful renewal, and the placeholder attempts to renew the lease every 60 seconds for 4.5 minutes before giving up.

Interface Script	applyDone.py (new)
Purpose	notify scheduler that file-system changes have been persisted to home nodes
Arguments	<placeholder id> <server-side Trellis directory> <queue> <exit code (0=success)>
Exit Code	0 = success; anything else = retry
Output	None

Table 4.5: **Placeholder Interface: Commit Done**

library transparently.

4.4 Concluding Remarks

We have described the implementation of the redundant execution feature, as designed in Chapter 3. The Trellis File System library has been modified to use a new cache manager interface, with implicit and explicit write-back policies. The scheduler scripts and database schema, along with the zero-infrastructure placeholder, have been extended to support the new job states and state transitions. The zero-infrastructure placeholder leverages TFS' explicit write-back policy to control when file-system modifications are propagated to home nodes. Additionally, we have created two new indirect ways of accessing the Trellis File System: Trellis Samba (described in Section 3.1.7), and the TFS Wrapper. In the following chapter, we leverage this implementation to quantify the performance impact of redundant execution.

In the future, it would be beneficial to extend support for redundant execution to other placeholder types (not just zero infrastructure), allowing the feature to work with batch scheduling systems.

Chapter 5

Empirical Evaluation

Having described our prototype system, we proceed to measure its performance and fault tolerance in the context of the Proteome Analyst application. Empirical evaluation provides evidence that redundant execution improves performance and fault tolerance and introduces negligible overheads for an important class of workloads. Following this evaluation, the final section concludes. We show:

1. Redundant execution allows Trellis to transparently recover from placeholder failures without the need for user intervention (Section 5.3).
2. Redundant execution allows Trellis to maintain high performance when some nodes are slow (Section 5.4).
3. Our system introduces negligible overheads for medium- or large-grain CPU-intensive workloads (Section 5.5.1). For fine-grain CPU-intensive workloads, overheads become significant (Section 5.5.2), but may be partially mitigated by reducing secure-shell connection overheads (Section 5.5.3) or by batching jobs together (Section 5.5.4).

This evaluation is targeted towards a specific class of applications: high-performance-computing workloads comprised of independent, CPU-dominated jobs that communicate via the file system. A discussion of other types of workloads follows in Section 5.6.

5.1 Evaluation Methodology

The Proteome Analyst application was used to run HMMER (introduced in Section 3.1.4) against subsets of the *Mycoplasma pneumoniae* proteome. This proteome consists of the 687 proteins in *Mycoplasma pneumoniae* which have Uniprot entries [16]. By varying experimental parameters and measuring the resulting overheads and overall execution time (makespan), we characterize the performance of five file-system access strategies.

Unless otherwise noted, all data points are averages of at least five runs, the $\pm$ values are standard deviation, secure-shell connection sharing is used (see Section 5.5.3 for details), and the batching

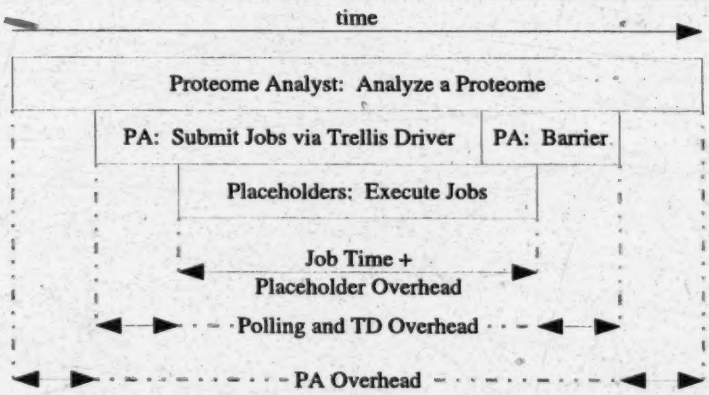


Figure 5.1: **Execution Time Breakdown – PA Analyzes a Proteome**

factor (i.e., the number of `hmmpfam` or `hmmsearch` commands per job) is two. Performance comparisons are validated via the t test at the 5% confidence level.

Tables 5.1, 5.2, and 5.3 detail the various overheads and how they are measured. As PA is used to analyze proteomes, execution time is measured and divided into four categories, shown visually in Figure 5.1:

1. **PA Overhead.** Time spent in PA before jobs are submitted to the scheduler and after those jobs complete.
2. **Polling and Trellis Driver (TD) Overhead.** Time PA spends submitting jobs and waiting for their completion while no placeholder has assigned work.
3. **Placeholder Overheads.** Average time a placeholder spends on tasks other than job execution. These overheads are further broken down in Tables 5.2 and 5.3.
4. **Job Time.** Average time a placeholder spends executing jobs.

Performance data is gathered from three sources:

1. **PA Log File.** PA's log file includes overall makespan as well as the time spent in the job submission loop and subsequent barrier.
2. **Scheduler's Job Database.** The job database includes a timestamped event for each time a placeholder is assigned or completes a job.
3. **Placeholder Log Files.** The placeholder log files are most detailed of all, having been extended to include a start and stop timestamp for each logical operation.

Overhead	Description	How to Compute
PA	Pre- and post-processing in Proteome Analyst: 1. input file creation 2. output file parsing 3. database access to store results 4. transformation of `hmmsearch` output to the `hmmpfam` format, if applicable	1. "PA Overhead" = "makespan [PA log]" – "duration of job submission loop and barrier [PA log]"
Polling and TD	Communication delays between PA and placeholders: 1. Trellis Driver overheads (believed negligible) 2. polling delay for first placeholder to request work 3. polling delay to detect completion of last job	1. "duration where placeholders are doing useful work" = "time when last job is marked complete [job database]" – "time when first job is assigned to a placeholder [job database]" 2. "Polling and TD Overhead" = "duration of job submission loop and barrier [PA log]" – "duration where placeholders are doing useful work"
Placeholder	Placeholder time not spent on job execution. See Tables 5.2 and 5.3 for a detailed breakdown.	1. "average placeholder job time" = $\sum_{i \in PH}$("time placeholder i spent executing jobs [PH log]") / "number of placeholders" Note: Job time does not include redundant executions of jobs at the end of the workload. It does include redundant executions in the middle of the workload as well as executions that fail; these are not significant for the benchmarks under test. 2. "Placeholder Overhead" = "duration where placeholders are doing useful work" – "average placeholder job time"

Table 5.1: **High-level Overheads.** Execution time is divided between job time and these categories in the high-level performance graphs. Variables in the "How to Compute" column are expressed as "textual description [data source]".

Overhead	Description	How to Compute
Get Job	Average time placeholders spend requesting work. For each job execution: 1. synchronous message to scheduler 2. database access	"Get Job Overhead" = $\sum_{i \in PH}$("time placeholder i spent requesting work [PH log]") / "number of placeholders"
Notify Job Complete	Average time placeholders spend notifying the scheduler that jobs are complete. For each job execution: 1. synchronous message to scheduler 2. database access	"Notify Job Complete Overhead" = $\sum_{i \in PH}$("time placeholder i spent notifying the scheduler that jobs are complete [PH log]") / "number of placeholders"
Commit Changes	Average time placeholders spend committing file-system changes (TFS-EXPL and SAMBA-EXPL only). Each job execution is either committed or rolled back. For commits: 1. data transfer to home nodes 2. file-system metadata changes on home nodes Other strategies include these steps in job execution.	"Commit Changes Overhead" = $\sum_{i \in PH}$("time placeholder i spent committing file-system changes [PH log]") / "number of placeholders"
Notify Commit Complete	Average time placeholders spend notifying the scheduler that commits are complete (TFS-EXPL and SAMBA-EXPL only). Following a commit: 1. synchronous message to scheduler 2. database access	"Notify Commit Complete Overhead" = $\sum_{i \in PH}$("time placeholder i spent notifying the scheduler that commits are complete [PH log]") / "number of placeholders"
Rollback Changes	Average time placeholders spend rolling back file-system changes (TFS-EXPL and SAMBA-EXPL only). When a job execution completes after a redundant one, the local store of pending changes is discarded.	"Rollback Changes Overhead" = $\sum_{i \in PH}$("time placeholder i spent rolling back file-system changes [PH log]") / "number of placeholders"
Sleep (Work Starvation)	Average time placeholders spend sleeping when there is no available work. Causes include: 1. **Slow producer.** The job producer (PA, Trellis Driver) falls behind. 2. **Dependencies.** All remaining jobs depend on running ones. This is not relevant to the workloads presented here.	"Sleep Overhead" = $\sum_{i \in PH}$("time placeholder i spent sleeping [PH log]") / "number of placeholders"
Placeholder Script	Average time placeholders spend on everything not accounted for above (e.g., flow control or writing debug output).	"Placeholder Script Overhead" = $\sum_{i \in PH}$("time placeholder i spent between timed operations [PH log]") / "number of placeholders"

Table 5.2: **Placeholder Overheads Part I: Overheads Based on the Placeholder Logs.** Placeholder overhead is divided into these categories (and those from Table 5.3) in the placeholder overhead graphs. Variables in the "How to Compute" column are expressed as "textual description [data source]".

Overhead	Description	How to Compute
Staggered Placeholder Start Times	Average time placeholders are idle at the beginning of the workload. It is influenced by: 1. differences in when placeholders were launched 2. placeholder polling interval (60 seconds) before work is available 3. potential bottleneck adding jobs to the job database at the beginning of the workload	1. "placeholder i start time" = "time when first job is assigned to placeholder i [job database]" 2. "first placeholder start time" = $\min_{i \in PH}$("placeholder i start time") 3. "Staggered Placeholder Start Times Overhead" = $\sum_{i \in PH}$("placeholder i start time" − "first placeholder start time") / "number of placeholders"
Staggered Placeholder End Times	Average time placeholders are idle at the end of the workload, including: 1. waiting for other placeholders to complete, either because they are slow or simply due to imperfect load balancing 2. redundantly executing jobs which are ultimately completed by another placeholder first	1. "placeholder i end time" = "time when placeholder i last marks a job complete [job database]" 2. "last placeholder end time" = $\max_{i \in PH}$("placeholder i end time") 3. "Staggered Placeholder End Times Overhead" = $\sum_{i \in PH}$("last placeholder end time" − "placeholder i end time") / "number of placeholders"
Placeholder Correction	A small negative time to correct for activities counted both from the placeholder and scheduler perspectives: 1. the average duration between a placeholder initiating its first successful work request and that request being logged by the scheduler 2. the average duration between the scheduler logging the last work completion notification for a placeholder and control returning to that placeholder Clock skew between the scheduler and placeholder nodes, if significant, would also influence this value.	1. "placeholder i duration as seen by scheduler" = "placeholder i end time" − "placeholder i start time" 2. "placeholder i duration as seen by placeholder" = "time placeholder i completes notifying the scheduler about its last complete job [PH log]" − "time immediately preceding placeholder i's first successful request for work [PH log]" 3. "Placeholder Correction" = $\sum_{i \in PH}$("placeholder i duration as seen by scheduler" − "placeholder i duration as seen by placeholder") / "number of placeholders"

Table 5.3: **Placeholder Overheads Part II: Overheads Based on the Job Database.** Placeholder overhead is divided into these categories (and those from Table 5.2) in the placeholder overhead graphs. Variables in the "How to Compute" column are expressed as "textual description [data source]".

Figure 5.2 compares the two ways PA can compute HMMER results for a set of proteins [4]. HMMER tries to match proteins with protein families, which are modeled by hidden Markov models (HMMs). In the context of PA, users provide whole or partial proteomes (groups of proteins) to be analyzed together. In this case, HMMER needs to compare each protein with each HMM from a static database. This task can be divided in many ways, but the simplest are as follows:

1. Use the `hmmpfam` tool to compare individual proteins with all HMMs (Figure 5.2a).

2. Use the `hmmsearch` tool to compare individual HMMs with all proteins (Figure 5.2b).

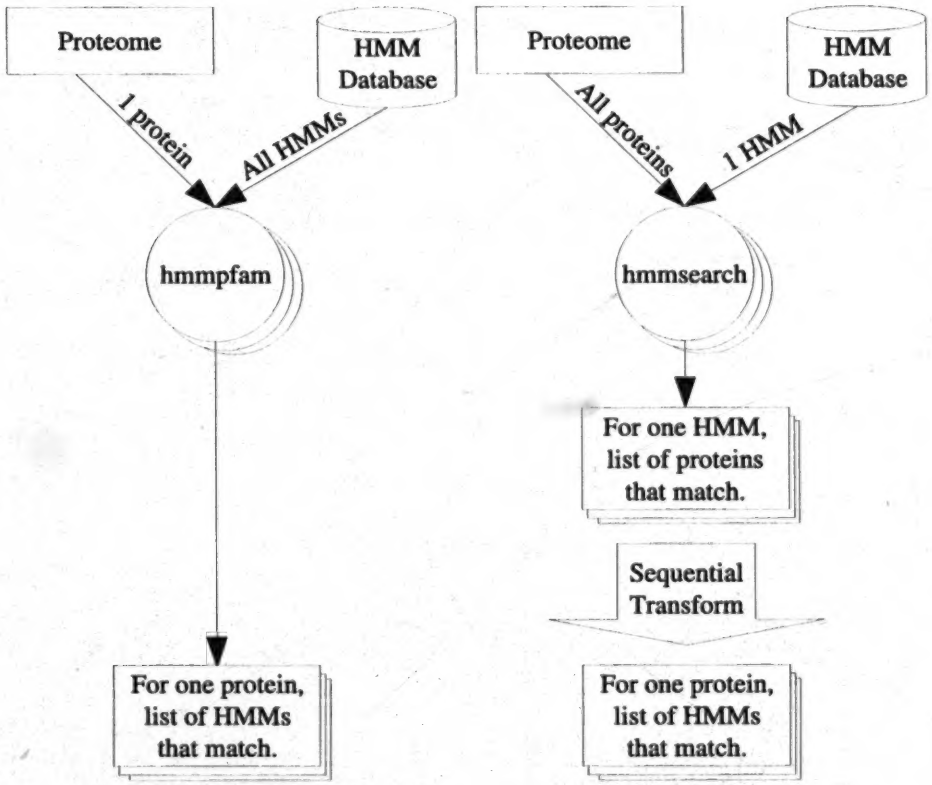
Each execution of `hmmpfam` or `hmmsearch` causes one input file to be read and one output file to be written. In addition, one of a number of static read-only files is read, and these are replicated so as to be local to each placeholder. The strategy for accessing files during job execution impacts performance, deployment options, and effort required to port applications.

We compare five file-system access strategies: NFS, TFS-IMPL, TFS-EXPL, SAMBA-IMPL, and SAMBA-EXPL. The NFS strategy stores shared data on a central NFS server. The other four strategies indirect file-system access via the TFS library, with data ultimately residing on the node where PA executes. Within these four strategies, there are two access mechanisms (TFS and SAMBA, see Table 5.4) and two write-back policies (implicit and explicit, see Section 3.2.3); the strategies are named by concatenating an access mechanism with a write-back policy.

We are interested in the performance of the explicit write-back strategies (TFS-EXPL and SAMBA-EXPL) relative to the others in various scenarios. These strategies isolate file-system accesses during job execution, allowing redundant execution, with the goal of reducing the impact of slow or failed nodes. The NFS strategy is a good comparison point because NFS is a ubiquitous, simple, and efficient way to share data in a LAN context. The implicit write-back strategies (TFS-IMPL and SAMBA-IMPL) are good comparison points because (a) they allow sharing of data across wide area networks and administrative domains, and (b) they are based on the same TFS library code, allowing us to evaluate the impact of redundant execution in isolation. The NFS, SAMBA-IMPL, and SAMBA-EXPL strategies generally allow the use of unmodified applications, while the TFS-IMPL and TFS-EXPL strategies generally require applications to be modified to access the file system via the TFS library.

5.2 Experimental Platform

There are three experimental platforms: one including two administrative domains (Experimental Platform A, Table 5.5), another with two computers of different speeds (Experimental Platform B, Table 5.6), and a homogeneous cluster (Experimental Platform C, Table 5.7). These platforms are configured as described in Figures 3.1 (for NFS), 3.2 (for TFS-IMPL and TFS-EXPL), and 3.3 (for SAMBA-IMPL and SAMBA-EXPL). Only Experimental Platform B supports Samba.



(a) hmmpfam

(b) hmmsearch

	hmmpfam	**hmmsearch**
strategy	Compare individual proteins against all protein families.	Compare individual protein families against all proteins.
granularity	medium (minutes)	small (seconds or less for hundreds of proteins), grows linearly with proteome size, highly variable by protein family
number of jobs (before batching), input files, and output files	same as the number of proteins	same as the number of protein families (7502)
post-processing	Populate database with contents of output files.	Transform hmmsearch-style output to hmmpfam-style output. Populate database with contents of transformed output files.

(c) Comparison Points

Figure 5.2: **Using HMMER to Match Proteins to Protein Families: hmmpfam versus hmmsearch.** Figures (a) and (b) are based on RAMMER [21].

Mechanism	Description
TFS	A wrapper program (Section 4.3) linked to the TFS library mediates access to the input and output files; the input file is passed to the HMMER executable on standard input and the output file is populated from standard output. This is representative of applications which are modified to use the TFS library directly.
SAMBA	An extended Samba server (Section 3.1.7) linked to the TFS library transparently provides access to remote files; from the perspective of the HMMER executable, it is directly accessing the input and output files. In general, this mechanism can be used to support unmodified applications.

Table 5.4: **File-system Access Mechanisms**

Host	calais	cleardale	NotScout
CPU	2.2 GHz AMD 64	1 GHz Pentium III	Dual 2.7 GHz G5
Memory	1 GB	256 MB	2 GB
OS	Linux 2.6.9	Linux 2.4.20	Mac OS X 10.4
Role	PA, scheduler, 1 placeholder	1 placeholder	2 placeholders
Location	local		remote, same city
Network	100 Mbit/s between local hosts; DSL between local and remote hosts		

Table 5.5: **Experimental Platform A – Three Nodes in Two Administrative Domains**

Host	lamont	cleardale	brule
CPU	2.2 GHz AMD	1 GHz Pentium III	dual 1.8 GHz AMD
Memory	1 GB	256 MB	2 GB
OS	Linux 2.4.25	Linux 2.4.20	Linux 2.6.11
Role	PA, scheduler, Trellis Samba, 1 placeholder, CVS server (shared)	1 placeholder	NFS server (shared)
Location	local		
Network	100 Mbit/s		

Table 5.6: **Experimental Platform B – Two Heterogeneous Nodes with Samba**

Host	jasper-xx	jasper-xx (4x or 8x)	brule
CPU	dual 1.5 GHz AMD		dual 1.8 GHz AMD
Memory	1.5 GB		2 GB
OS	Linux 2.4.18		Linux 2.6.11
Role	PA, scheduler	2 placeholders each	NFS server (shared)
Location	local		
Network	1 Gbit/s		

Table 5.7: **Experimental Platform C – Cluster**

5.3 Fault Tolerance and Multiple Administrative Domains

In the first experiment, we evaluate our system with placeholders spanning multiple administrative domains, with and without artificial placeholder-node failure. Experimental Platform A was used (Table 5.5), providing four placeholders in two administrative domains. File-system access was mediated with the TFS wrapper, and secure-shell connection sharing (Section 5.5.3) was not used. 100% of the proteome (687 proteins) was analyzed with `hmmpfam`.

In a set of single runs, the system took 4 hours 53 minutes, and 4 hours 58 minutes to execute with the TFS-IMPL and TFS-EXPL strategies, respectively. This demonstrates that the TFS-EXPL strategy, which allows for redundant execution, has a similar performance to the traditional one. We explore this result more thoroughly in Section 5.5.1. This experiment also demonstrates that the system can do real work while accessing files across multiple administrative domains.

In a third run, using the TFS-EXPL strategy, NotScout was rebooted after two hours while running jobs, leaving the computation at that point. The workload completed successfully after 9 hours 46 minutes. The fact that the workload completed at all demonstrates that redundant execution allows the system to recover automatically from failed placeholders that are executing jobs. Based on the job execution rates of each machine, we expect that the workload would take much longer to complete than 9 hours 46 minutes without NotScout's contribution during the first two hours. The remainder of this chapter focuses on overheads introduced by our redundant execution approach.

5.4 Slow Nodes

To demonstrate that our system maintains high performance when faced with slow nodes, we artificially slow one of four cluster nodes by five or ten times. We use `hmmpfam` with the four node cluster configuration to analyze 343 proteins. Figure 5.3 summarizes the results; for details, compare Figures 5.4, 5.6, and 5.8 and Figures 5.5, 5.7, and 5.9.

When no nodes are slowed, all three strategies perform about the same; compare makespans in Figure 5.4, e.g. 5592 $\pm$ 25 s for TFS-IMPL and 5622 $\pm$ 48 s for TFS-EXPL. When one node is slowed by approximately five times, the TFS-EXPL strategy uses redundancy to avoid waiting for the slow node at the end of the workload, and thus gains an advantage, e.g. 7604 $\pm$ 269 s for TFS-IMPL versus 7031 $\pm$ 103 s for TFS-EXPL (Figure 5.6). This advantage grows when the node is further slowed to a factor of approximately ten times. Although the TFS-EXPL strategy successfully mitigates the slow node's negative effect on the workload tail, makespan is still higher than when the node was not slowed. This is because slowing the node reduces the overall throughput of the cluster.

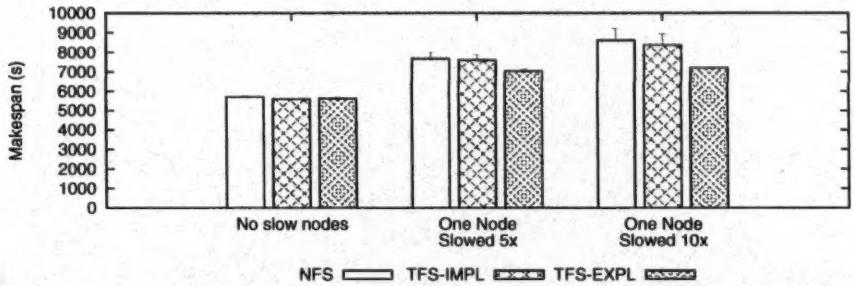
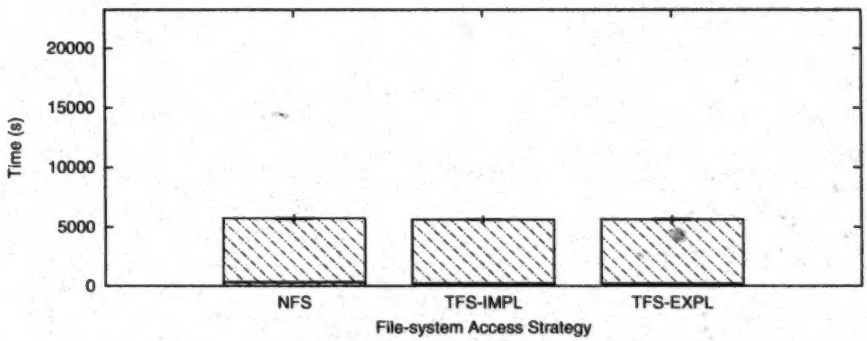


Figure 5.3: **Impact of Redundant Execution with Slow Nodes: hmmpfam, 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. See Figures 5.4, 5.6, and 5.8 for details. When one node is artificially slowed, the TFS-EXPL strategy outperforms the others because it can use redundancy to avoid waiting for the slow node at the end of the workload. This effect is larger when the node is slowed more. All strategies, including TFS-EXPL, decrease in performance as the node is increasingly slowed. This is partly due to a reduction in the cluster's overall throughput.

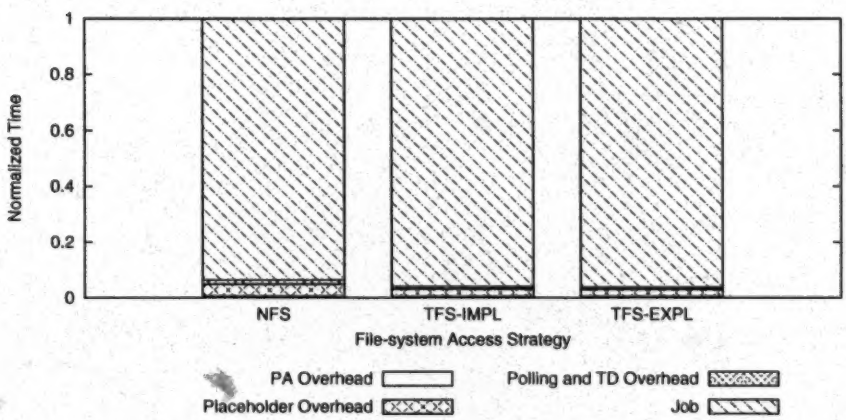
5.5 Overheads

In this section, we leverage the two HMMER execution strategies (Figure 5.2) to evaluate the performance of our system for medium- and fine-grain workloads. When using `hmmpfam`, each job reads and writes two small files (typically ≤2 kB input, ≤25 kB output), and spends minutes in CPU-bound computation; this is an example of a medium-grain workload. When using `hmmsearch` with 343 proteins, each job also reads and writes two small files (typically 150 kB input, ≤8 kB output), but many jobs complete in less than a second; this is an example of a fine-grain workload. The granularity of the `hmmsearch` workload can be increased by increasing the batching factor (e.g., from 2 to 32), but the ratio of file accesses to computation time remains high. Another significant difference is that the process of converting hmmsearch-style output to hmmpfam-style output – required for PA – is sequential, which suggests that the `hmmpfam` approach will scale better with the number of CPUs.

Figure 5.10 shows the relative performance of using `hmmpfam` versus `hmmsearch` to compute HMMER results for 343 proteins. In general, the medium-grain workload (i.e., `hmmpfam`) is much less sensitive to file-system access strategy, where the fine-grain workload (i.e., `hmmsearch`) highlights significant overheads imposed by the TFS library and the explicit write-back policy. In previous work, using `hmmsearch` with PA was shown to be as much as three times more efficient than `hmmpfam` for large proteomes with a high batching factor, four CPUs, and shared NFS storage [4]. In the figure, we can see that even with a small proteome and small batching factor, `hmmsearch` completes more quickly. In this work, we are more interested in the impact of the explicit write-back strategies on medium- and fine-grain workloads than the relative advantages of `hmmpfam` or `hmmsearch`. It is worth noting, however, that real problems often may be solved in different ways, resulting in different workloads with different performance characteristics.



(a) High-level Overheads in Context. Legend shared with Figure (b).

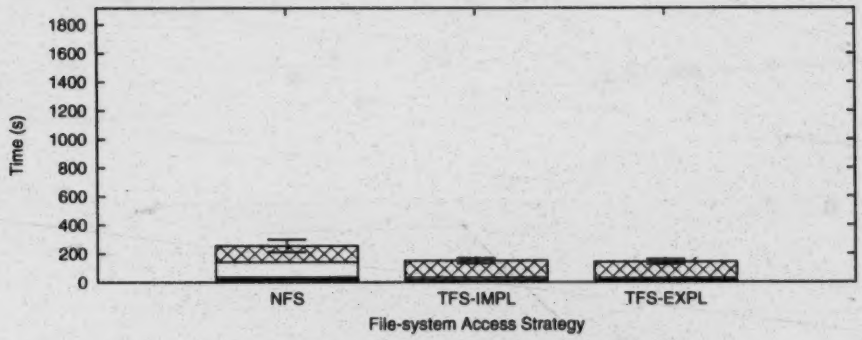


(b) High-level Overheads in Context, Normalized

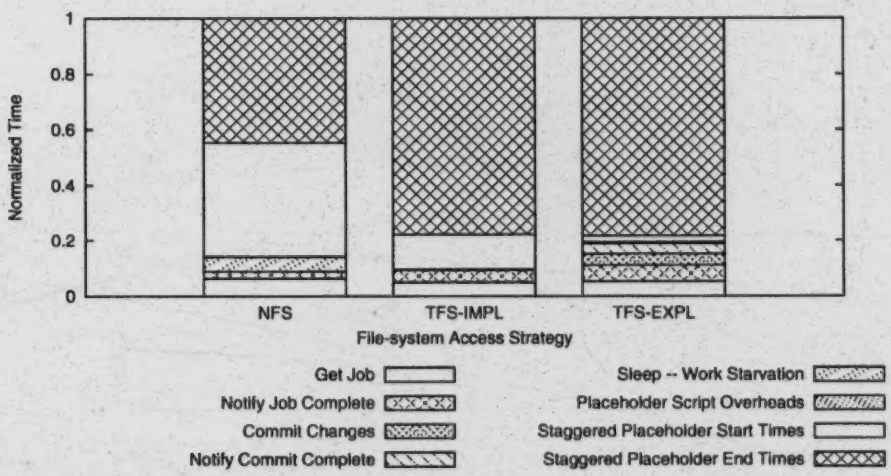
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	13	11	10
PA Overhead (s)	33 ± 2	28 ± 1	28 ± 0
Placeholder Overhead (s)	247 ± 38	147 ± 19	140 ± 18
Polling and TD Overhead (s)	83 ± 11	48 ± 9	45 ± 15
Job (s)	5341 ± 1	5369 ± 11	5409 ± 48
Job + Commit (s)	5341 ± 1	5369 ± 11	5414 ± 49
Makespan (s)	5704 ± 41	5592 ± 25	5622 ± 48

(c) High-level Overheads in Context, Source Data

Figure 5.4: **Performance Breakdown: hmmpfam (medium grain), 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. The medium-grain `hmmpfam` jobs and large workload result in small overheads and similar performance for the three strategies. The slight disadvantage for NFS can be traced to the "Staggered Placeholder Start Times" and "Sleep – Work Starvation" overheads (see Figure 5.5f), which are believed to be caused by database contention during job submission.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).

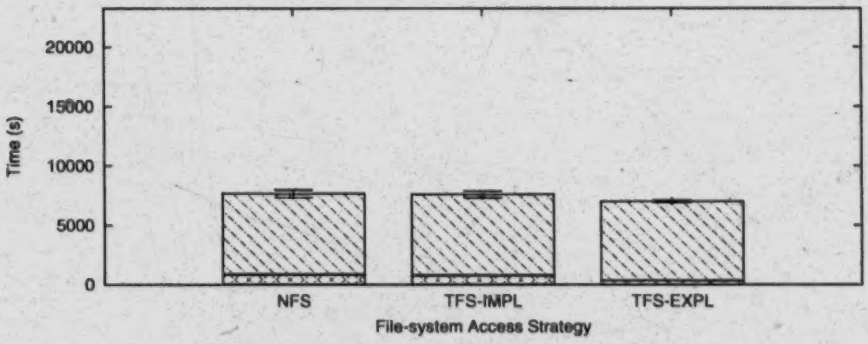


(e) Placeholder Overhead Detail, Normalized

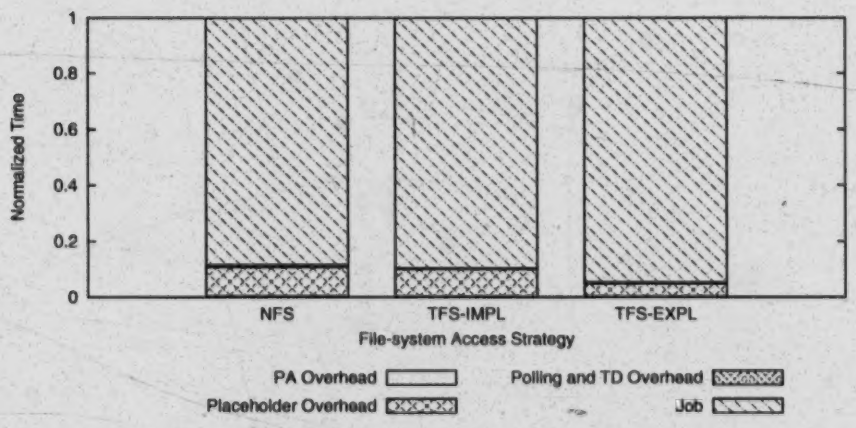
	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	16 ± 10	7 ± 2	7 ± 2
Notify Job Complete (s)	7 ± 2	7 ± 2	8 ± 5
Commit Changes (s)	0 ± 0	0 ± 0	6 ± 1
Notify Commit Complete (s)	0 ± 0	0 ± 0	5 ± 1
Sleep – Work Starvation (s)	13 ± 9	0 ± 0	0 ± 0
Placeholder Script Overheads (s)	1 ± 0	0 ± 0	1 ± 0
Staggered Placeholder Start Times (s)	105 ± 39	19 ± 14	3 ± 5
Staggered Placeholder End Times (s)	114 ± 21	117 ± 15	112 ± 16
Placeholder Correction (s)	-9 ± 8	-3 ± 2	-3 ± 2
Total (s)	247 ± 38	147 ± 19	140 ± 18

(f) Placeholder Overhead Detail, Source Data

Figure 5.5: **Placeholder Overheads: hmmpfam (medium grain), 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. Overheads are minimal and dominated by staggered placeholder start and end times. As mentioned in Figure 5.4, NFS experiences higher "Staggered Placeholder Start Times" and "Sleep – Work Starvation" overheads.



(a) High-level Overheads in Context. Legend shared with Figure (b).

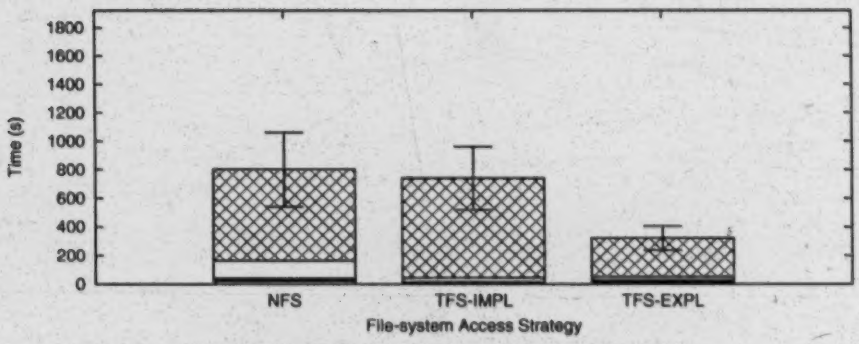


(b) High-level Overheads in Context, Normalized

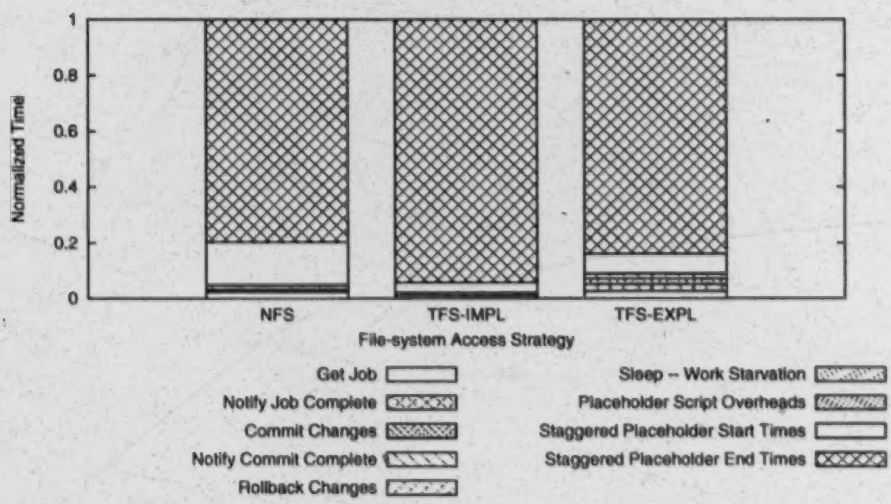
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	23	24	23
PA Overhead (s)	34 ± 1	28 ± 1	28 ± 0
Placeholder Overhead (s)	790 ± 261	737 ± 222	318 ± 84
Polling and TD Overhead (s)	71 ± 17	32 ± 19	32 ± 18
Job (s)	6779 ± 71	6807 ± 59	6652 ± 58
Job + Commit (s)	6779 ± 71	6807 ± 59	6659 ± 58
Makespan (s)	7675 ± 314	7604 ± 269	7031 ± 103

(c) High-level Overheads in Context, Source Data

Figure 5.6: **Performance Breakdown: hmmpfam (medium grain), 343 proteins, 4×2 cluster CPUs, one node slowed 5x.** Smaller numbers are better. The TFS-EXPL case, which benefits from redundant execution, completes sooner than the others due to reduced placeholder overheads.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).

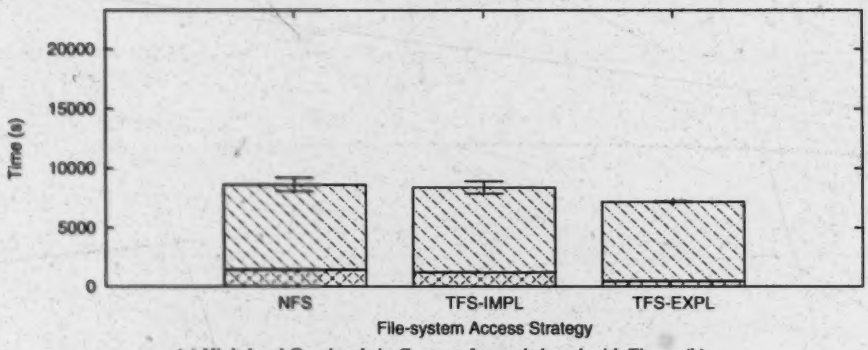


(e) Placeholder Overhead Detail, Normalized

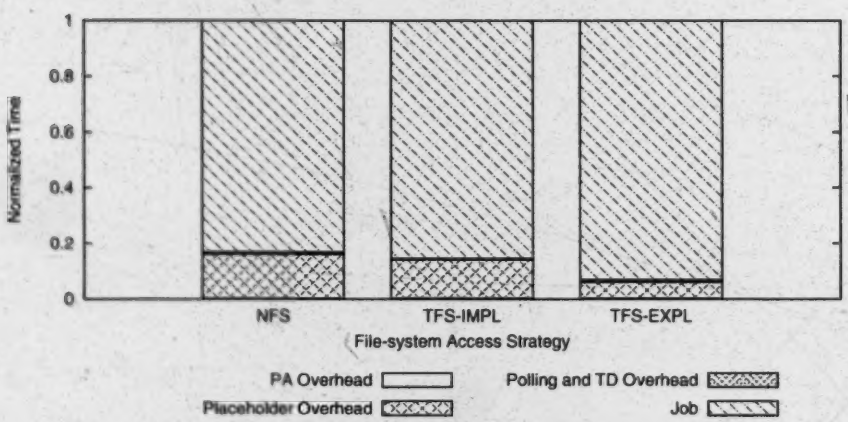
	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	19 ± 10	8 ± 4	8 ± 5
Notify Job Complete (s)	8 ± 2	7 ± 2	8 ± 4
Commit Changes (s)	0 ± 0	0 ± 0	6 ± 1
Notify Commit Complete (s)	0 ± 0	0 ± 0	5 ± 1
Rollback Changes (s)	0 ± 0	0 ± 0	0 ± 0
Sleep – Work Starvation (s)	12 ± 11	0 ± 0	0 ± 0
Placeholder Script Overheads (s)	1 ± 0	1 ± 0	1 ± 0
Staggered Placeholder Start Times (s)	124 ± 43	27 ± 6	23 ± 7
Staggered Placeholder End Times (s)	639 ± 250	697 ± 222	270 ± 86
Placeholder Correction (s)	-12 ± 8	-3 ± 3	-4 ± 5
Total (s)	790 ± 269	737 ± 222	318 ± 84

(f) Placeholder Overhead Detail, Source Data

Figure 5.7: **Placeholder Overheads: hmmpfam (medium grain), 343 proteins, 4×2 cluster CPUs, one node slowed 5x.** Smaller numbers are better. Placeholder overheads are dominated by the "Staggered Placeholder End Times" overhead; often, fast placeholders are forced to wait for the slow ones at the end of the workload. The TFS-EXPL strategy uses redundant execution to reduce this overhead. The large standard deviations for the NFS and TFS-IMPL strategies occur because sometimes the slow placeholders complete at just the right time and the fast ones do not have to wait.



(a) High-level Overheads in Context. Legend shared with Figure (b).

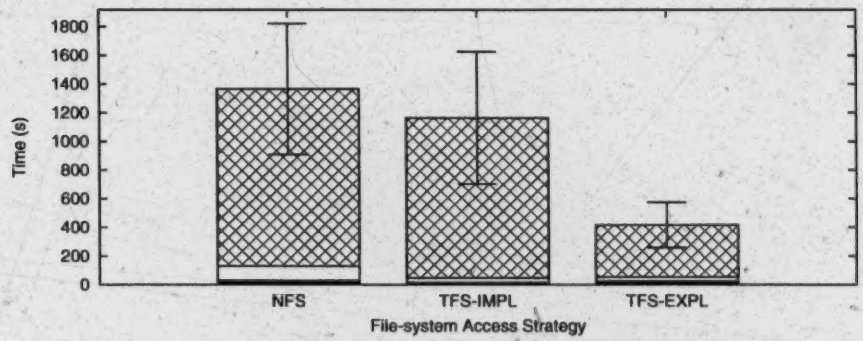


(b) High-level Overheads in Context, Normalized

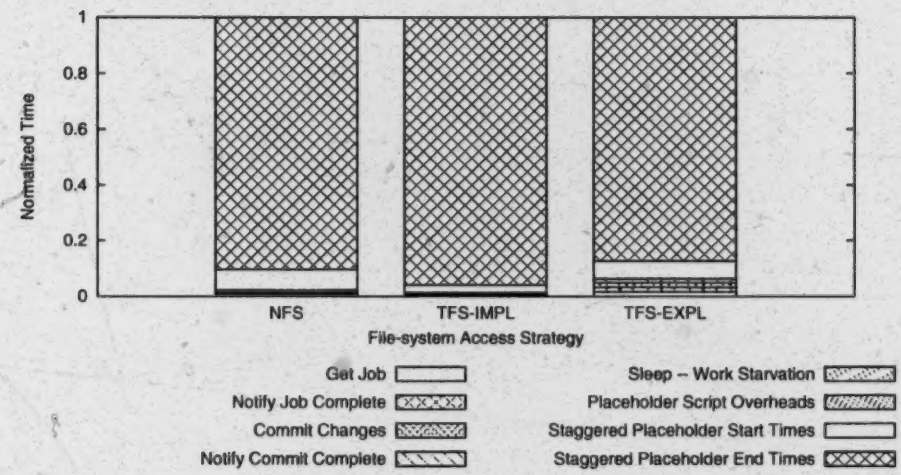
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	5	5	5
PA Overhead (s)	35 ± 2	28 ± 0	28 ± 1
Placeholder Overhead (s)	1360 ± 461	1161 ± 461	415 ± 159
Polling and TD Overhead (s)	66 ± 21	43 ± 37	62 ± 23
Job (s)	7167 ± 126	7165 ± 88	6697 ± 143
Job + Commit (s)	7167 ± 126	7165 ± 88	6704 ± 142
Makespan (s)	8627 ± 577	8398 ± 537	7202 ± 28

(c) High-level Overheads in Context, Source Data

Figure 5.8: **Performance Breakdown: hmmpfam (medium grain), 343 proteins, 4×2 cluster CPUs, one node slowed 10x.** Smaller numbers are better. Compared with Figure 5.6 (one node slowed only 5x), the TFS-EXPL strategy is a little bit slower (~200s) and the others are a lot slower (~750s).



(d) Placeholder Overhead Detail. Legend shared with Figure (e).

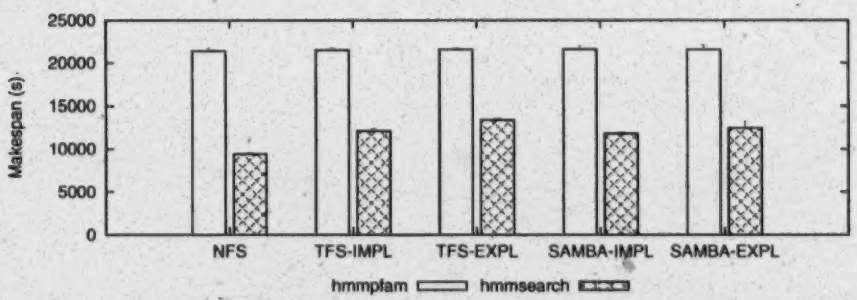


(e) Placeholder Overhead Detail, Normalized

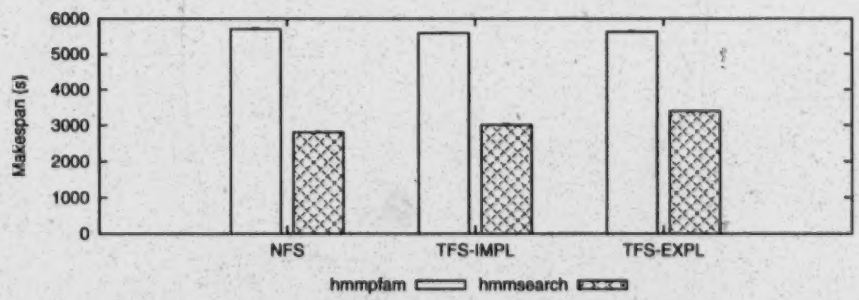
	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	12 ± 10	8 ± 5	7 ± 1
Notify Job Complete (s)	8 ± 3	7 ± 1	7 ± 2
Commit Changes (s)	0 ± 0	0 ± 0	7 ± 1
Notify Commit Complete (s)	0 ± 0	0 ± 0	5 ± 1
Sleep – Work Starvation (s)	11 ± 4	0 ± 0	0 ± 0
Placeholder Script Overheads (s)	1 ± 0	1 ± 0	1 ± 0
Staggered Placeholder Start Times (s)	101 ± 56	32 ± 6	26 ± 15
Staggered Placeholder End Times (s)	1235 ± 515	1117 ± 462	364 ± 156
Placeholder Correction (s)	-7 ± 8	-4 ± 6	-2 ± 2
Total (s)	1360 ± 461	1161 ± 461	415 ± 159

(f) Placeholder Overhead Detail, Source Data

Figure 5.9: **Placeholder Overheads: hmmpfam (medium grain), 343 proteins, 4×2 cluster CPUs, one node slowed 10x.** Smaller numbers are better. As with Figure 5.7 (one node slowed only 5x), the TFS-EXPL strategy is able to mitigate the "Staggered Placeholder End Times" overhead resulting from slow nodes.



(a) Two heterogeneous nodes. See Figures 5.11 and 5.15 for details.



(b) 4×2 cluster CPUs. See Figures 5.4 and 5.17 for details.

Figure 5.10: **Relative Performance: hmmpfam versus hmmsearch, 343 proteins.** Smaller numbers are better. The TFS library and explicit write-back strategies introduce few overheads for the medium-grain `hmmpfam` workload, and significant overheads for the fine-grain `hmmsearch` workload. The `hmmsearch` workload is more efficient overall.

5.5.1 HMMER via hmmpfam (Medium-Grain Jobs)

Here, we use `hmmpfam` to show that the overhead of redundant execution is negligible for medium-grain workloads. The following experiments evaluate 343 proteins with `hmmpfam` on Experimental Platform B (two heterogeneous nodes) and Experimental Platform C (cluster with 4 or 8 placeholder nodes).

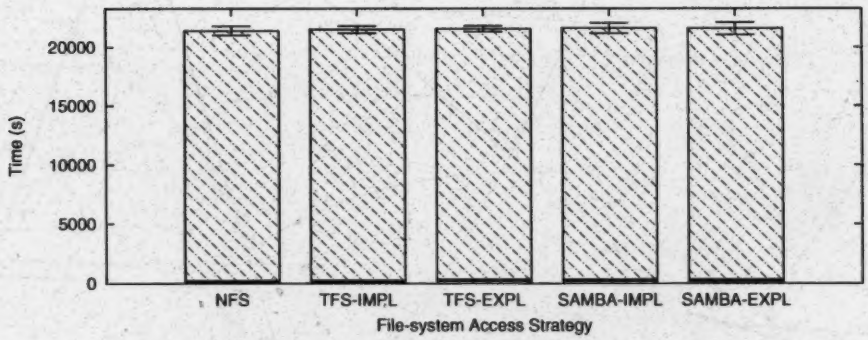
The results for two heterogeneous nodes are shown in Figures 5.11 and 5.12. For this medium-grain workload, overheads are small (1-2%), and the majority of the makespan is spent executing jobs. For example, the SAMBA-IMPL strategy spends 21247 $\pm$ 379 s executing jobs out of a total 21610 $\pm$ 432 s (Figure 5.11c). More notably, makespans for the five strategies are statistically indistinguishable; any overheads introduced by the TFS library or the explicit write-back policy are negligible in this case.

The results for the four-node cluster configuration are shown in Figures 5.4 and 5.5. The results for the eight-node cluster configuration are shown in Figures 5.13 and 5.14. Here, NFS performs worse than the other strategies due to higher placeholder overheads. In particular, NFS has higher "Staggered Placeholder Start Times" and "Sleep – Work Starvation" overheads, which are believed to be caused by database contention during job submission. In the job submission loop, Trellis Driver attempts to submit as many as 100 jobs concurrently, which can lead to deadlock / retry loops as the database code is not optimized for this use. It is unclear why this issue is significantly more pronounced for NFS than the other strategies, but it may be related to input-file accesses over NFS just prior to the job-submission loop. The TFS-IMPL and TFS-EXPL makespans remain very similar, e.g. 2935 $\pm$ 18 s for TFS-IMPL versus 2960 $\pm$ 40 s for TFS-EXPL in Figure 5.13c.

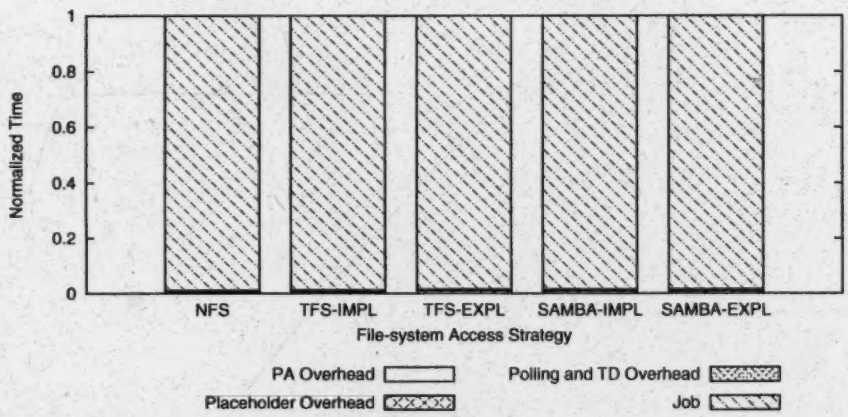
Comparing makespans for the four- and eight-node configurations, we see that `hmmpfam` scales well with the number of nodes. When doubling the number of CPUs, makespan is reduced by 46%, 48%, and 47% for NFS, TFS-IMPL, and TFS-EXPL respectively, with 50% being the ideal value. Although job time (including commit for TFS-EXPL) is reduced almost exactly to one half for each strategy, placeholder overheads make up the difference. It is clear that for a very large number of CPUs or a very small workload, placeholder overheads would become the dominant cost.

5.5.2 HMMER via hmmsearch (Fine-Grain Jobs)

Here, we use `hmmsearch` to evaluate the impact of redundant execution for fine-grain workloads. In contrast to the medium-grain workloads, overheads are larger, and in particular, the overhead of redundant execution is not negligible. The following experiments evaluate 343 proteins with `hmmsearch` on Experimental Platform B (two heterogeneous nodes) and Experimental Platform C (cluster with 4 placeholder nodes). The results for two heterogeneous nodes are shown in Figures 5.15 and 5.16. The results for the four-node cluster configuration are shown in Figures 5.17 and 5.18. We make a number of observations:



(a) High-level Overheads in Context. Legend shared with Figure (b).

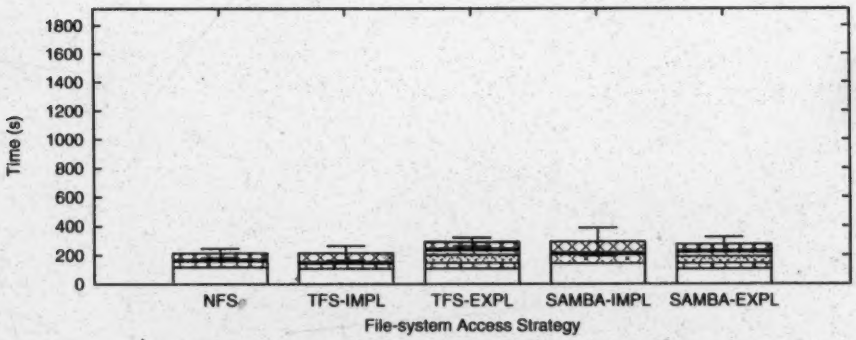


(b) High-level Overheads in Context, Normalized

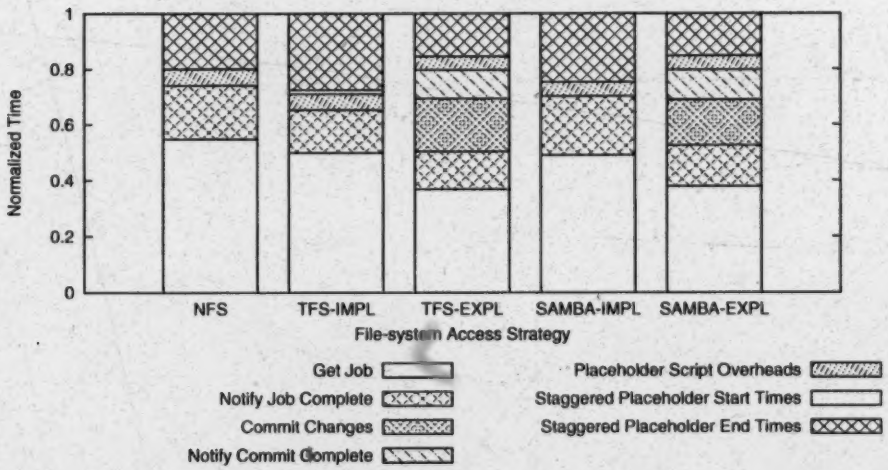
	NFS	TFS-IMPL	TFS-EXPL	SAMBA-IMPL	SAMBA-EXPL
Number of Samples	11	10	10	10	10
PA Overhead (s)	12 ± 2	12 ± 2	13 ± 1	14 ± 1	17 ± 6
Placeholder Overhead (s)	129 ± 40	139 ± 48	214 ± 23	209 ± 96	202 ± 38
Polling and TD Overhead (s)	134 ± 24	139 ± 27	120 ± 29	140 ± 19	143 ± 32
Job (s)	21125 ± 351	21208 ± 266	21213 ± 242	21247 ± 379	21214 ± 484
Job + Commit (s)	21125 ± 351	21208 ± 266	21269 ± 244	21247 ± 379	21259 ± 487
Makespan (s)	21401 ± 366	21498 ± 288	21561 ± 237	21610 ± 432	21576 ± 515

(c) High-level Overheads in Context, Source Data

Figure 5.11: **Performance Breakdown: hmmpfam (medium grain), 343 proteins, two heterogeneous nodes.** Smaller numbers are better. There are no statistically significant differences in makespan between the five strategies. Neither the TFS library, the explicit write-back policy, nor Trellis Samba introduce significant overheads or performance benefits versus NFS in this scenario.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).

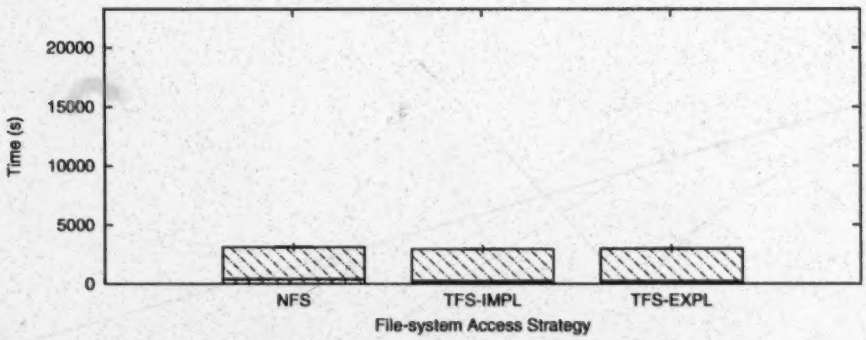


(e) Placeholder Overhead Detail, Normalized

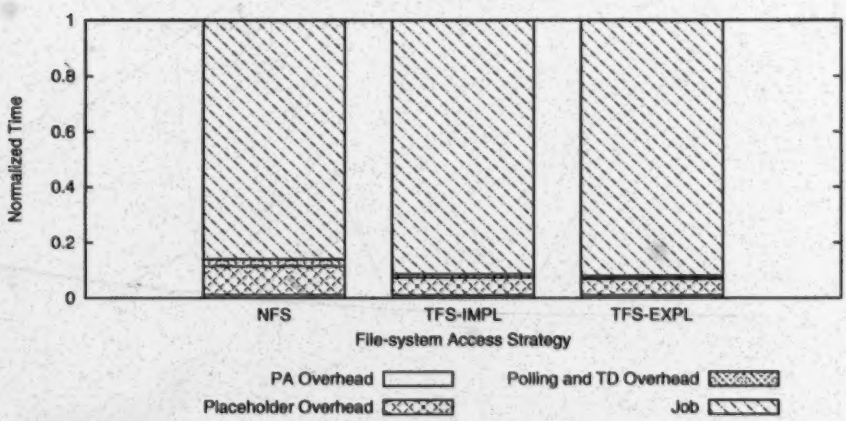
	NFS	TFS-IMPL	TFS-EXPL	SAMBA-IMPL	SAMBA-EXPL
Get Job (s)	117 ± 14	107 ± 29	107 ± 19	143 ± 50	104 ± 27
Notify Job Complete (s)	41 ± 6	33 ± 2	39 ± 3	62 ± 45	41 ± 3
Commit Changes (s)	0 ± 0	0 ± 0	55 ± 4	0 ± 0	45 ± 4
Notify Commit Complete (s)	0 ± 0	0 ± 0	30 ± 2	0 ± 0	30 ± 2
Placeholder Script Overheads (s)	13 ± 1	12 ± 1	14 ± 1	15 ± 5	14 ± 1
Staggered Placeholder Start Times (s)	0 ± 0	3 ± 10	0 ± 0	0 ± 0	0 ± 0
Staggered Placeholder End Times (s)	42 ± 36	59 ± 42	45 ± 22	72 ± 41	42 ± 29
Placeholder Correction (s)	-83 ± 14	-75 ± 29	-76 ± 19	-84 ± 11	-73 ± 27
Total (s)	129 ± 40	139 ± 48	214 ± 23	209 ± 96	202 ± 38

(f) Placeholder Overhead Detail, Source Data

Figure 5.12: **Placeholder Overheads: hmmpfam (medium grain), 343 proteins, two heterogeneous nodes.** Smaller numbers are better. The TFS-EXPL and SAMBA-EXPL strategies have a separate step to copy data back to the home node (commit changes), and an extra step to notify the scheduler when this is complete (notify commit complete).



(a) High-level Overheads in Context. Legend shared with Figure (b).

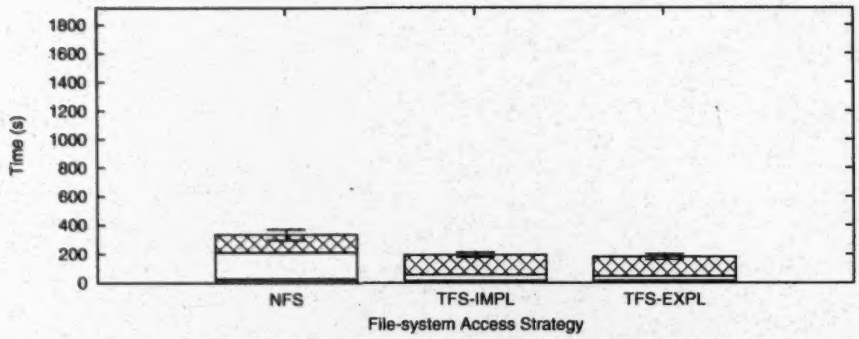


(b) High-level Overheads in Context, Normalized

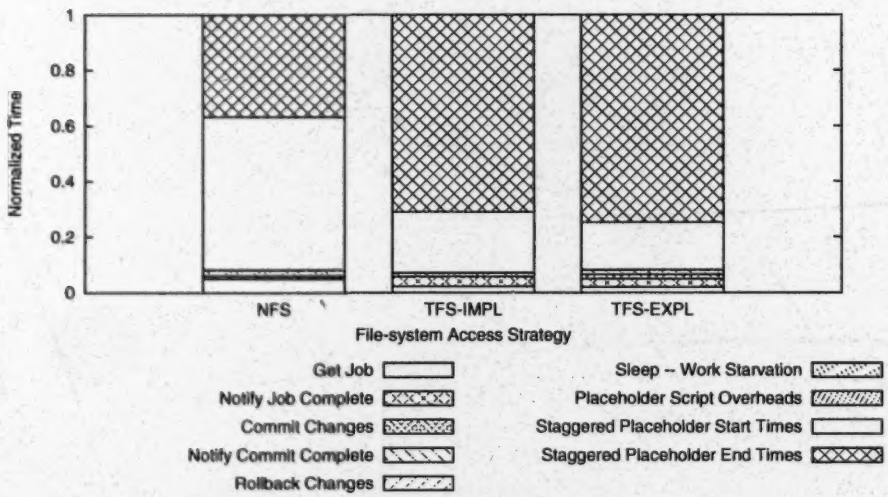
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	5	6	5
PA Overhead (s)	34 ± 1	29 ± 1	29 ± 0
Placeholder Overhead (s)	321 ± 26	191 ± 17	178 ± 19
Polling and TD Overhead (s)	78 ± 15	34 ± 21	29 ± 9
Job (s)	2672 ± 0	2681 ± 0	2724 ± 44
Job + Commit (s)	2672 ± 0	2681 ± 0	2727 ± 44
Makespan (s)	3105 ± 36	2935 ± 18	2960 ± 40

(c) High-level Overheads in Context, Source Data

Figure 5.13: **Performance Breakdown: hmmpfam (medium grain), 343 proteins, 8×2 cluster CPUs.** Smaller numbers are better. Compared with Figure 5.4 (half as many placeholder CPUs), job time (including commit for TFS-EXPL) is reduced to almost exactly half for each strategy. However, placeholder overheads are slightly higher. Overall, makespan is reduced by 46%, 48%, and 47% for NFS, TFS-IMPL, and TFS-EXPL respectively.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).



(e) Placeholder Overhead Detail, Normalized

	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	17 ± 15	4 ± 1	3 ± 1
Notify Job Complete (s)	3 ± 2	7 ± 3	5 ± 3
Commit Changes (s)	0 ± 0	0 ± 0	3 ± 0
Notify Commit Complete (s)	0 ± 0	0 ± 0	3 ± 0
Rollback Changes (s)	0 ± 0	0 ± 0	0 ± 0
Sleep – Work Starvation (s)	7 ± 5	2 ± 6	0 ± 0
Placeholder Script Overheads (s)	0 ± 0	0 ± 0	0 ± 0
Staggered Placeholder Start Times (s)	184 ± 45	42 ± 8	30 ± 1
Staggered Placeholder End Times (s)	123 ± 28	137 ± 19	134 ± 21
Placeholder Correction (s)	-12 ± 14	-1 ± 0	-1 ± 0
Total (s)	321 ± 26	191 ± 17	178 ± 19

(f) Placeholder Overhead Detail, Source Data

Figure 5.14: **Placeholder Overheads: hmmpfam (medium grain), 343 proteins, 8×2 cluster CPUs.** Smaller numbers are better. Compared with Figure 5.5 (half as many placeholder CPUs), placeholder overheads are slightly higher.

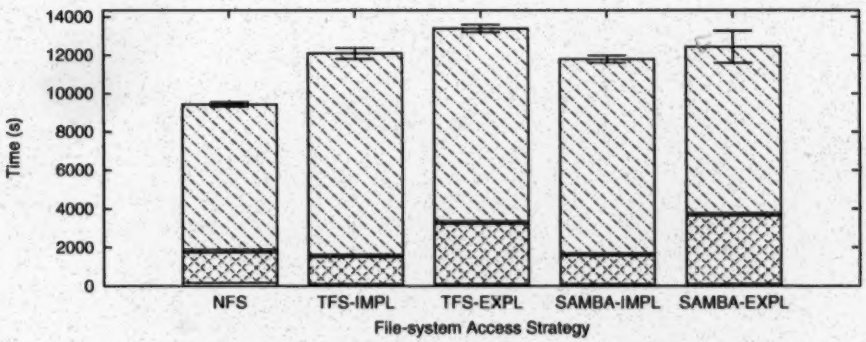
1. In contrast to the `hmmpfam` workloads, placeholder overheads are much larger; compare Figures 5.4 (`hmmpfam`) and 5.17 (`hmmsearch`), e.g. 147 ± 19 s for TFS-IMPL in Figure 5.4 versus 623 ± 37 s for TFS-IMPL in Figure 5.17. This results from the much larger number of `hmmsearch` jobs and their much shorter duration. Hence, more time is spent communicating with the scheduler to request work or signal job completion, overheads which are almost negligible with `hmmpfam`. This is a significant disadvantage for the explicit write-back strategies (TFS-EXPL and SAMBA-EXPL) which have an extra synchronous message to the scheduler for every job to signal completion of the commit step.

2. The NFS strategy performs much better than the others, primarily because it can transfer small files more quickly than the TFS library, which uses `scp`. This difference shows up in job time (including commit for TFS-EXPL and SAMBA-EXPL), where NFS has a clear advantage, e.g. 1715 ± 5 s for NFS versus 2188 ± 11 s for TFS-IMPL in Figure 5.17. For two heterogeneous nodes, the other strategies are as much as 42% slower overall, i.e. 13395 ± 193 s for TFS-EXPL versus 9436 ± 122 s for NFS in Figure 5.15. For the four-node cluster configuration, the other strategies are as much as 21% slower overall, i.e. 3394 ± 17 s for TFS-EXPL versus 2808 ± 48 s for NFS in Figure 5.17.

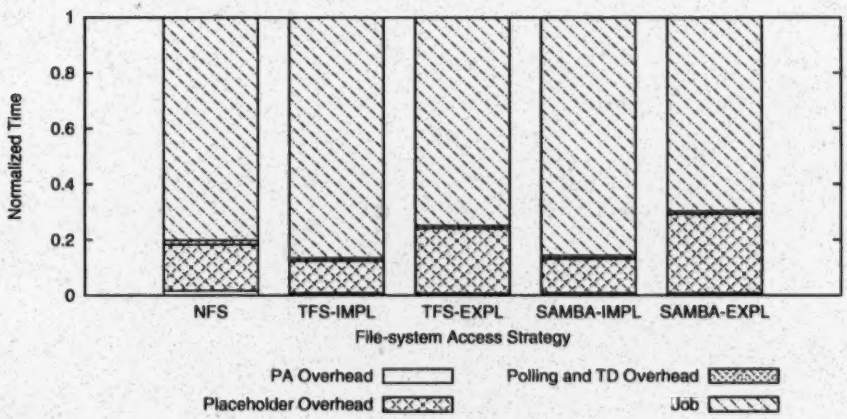
3. The SAMBA strategies perform better than their non-SAMBA counterparts, i.e. 11787 ± 186 s for SAMBA-IMPL versus 12090 ± 280 s for TFS-IMPL and 12436 ± 837 s for SAMBA-EXPL versus 13395 ± 193 s for TFS-EXPL in Figure 5.15. This is because Trellis Samba only needs to connect to the home node once to start the Trellis RPC agent – required for metadata operations – while the TFS-IMPL and TFS-EXPL strategies must do this for every execution of `hmmsearch`.

4. The explicit write-back strategies, which enable redundant execution, perform slightly worse than their implicit write-back counterparts; compare makespans for TFS-IMPL versus TFS-EXPL and SAMBA-IMPL versus SAMBA-EXPL in Figures 5.15 and 5.17, e.g. 3015 ± 37 s for TFS-IMPL versus 3394 ± 17 s for TFS-EXPL in Figure 5.17. This primarily results from the extra synchronous messages to the scheduler to notify completion of the commit step (the "Notify Commit Complete" overhead), e.g. 302 ± 8 s for TFS-EXPL in Figure 5.18.

5. The PA overhead is higher for NFS than for the other strategies because it has to access the NFS server to create the input files and read the output files, e.g. 294 ± 4 s for NFS versus 164 ± 2 s for TFS-IMPL in Figure 5.17c. This effect was also visible for the `hmmpfam` workloads, e.g. 33 ± 2 s for NFS versus 28 ± 1 s for TFS-IMPL in Figure 5.4c, but is more noticeable here because there are more files to read and write.



(a) High-level Overheads in Context. Legend shared with Figure (b).

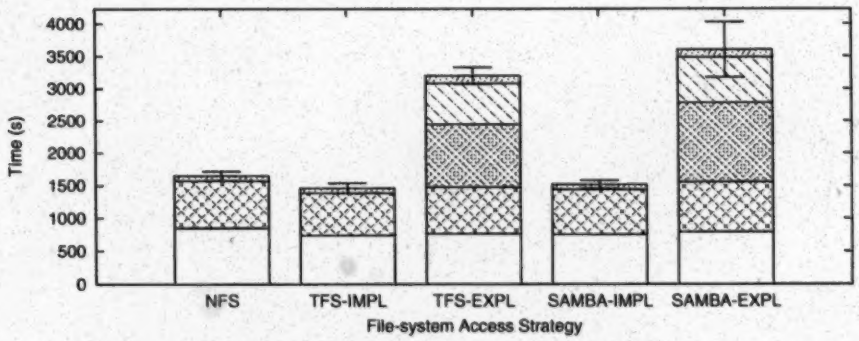


(b) High-level Overheads in Context, Normalized

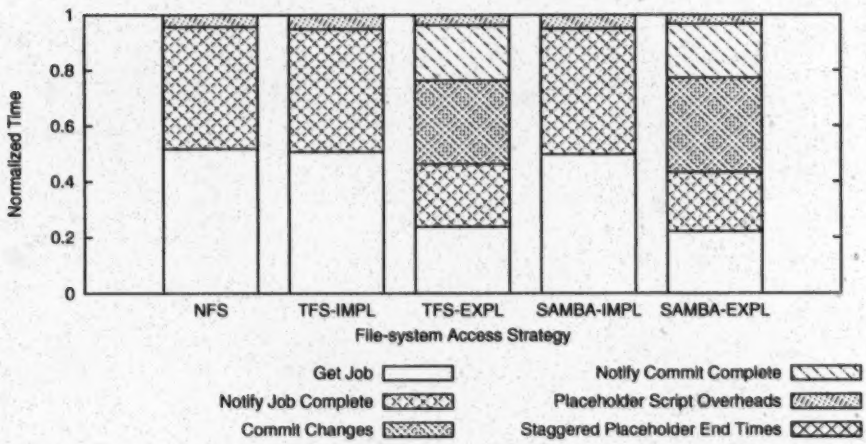
	NFS	TFS-IMPL	TFS-EXPL	SAMBA-IMPL	SAMBA-EXPL
Number of Samples	18	18	17	16	16
PA Overhead (s)	169 ± 18	101 ± 2	102 ± 3	102 ± 3	103 ± 2
Placeholder Overhead (s)	1570 ± 71	1404 ± 80	3133 ± 128	1462 ± 57	3542 ± 426
Polling and TD Overhead (s)	139 ± 14	129 ± 15	135 ± 13	128 ± 14	128 ± 13
Job (s)	7559 ± 50	10457 ± 203	10025 ± 74	10094 ± 168	8663 ± 457
Job + Commit (s)	7559 ± 50	10457 ± 203	10988 ± 123	10094 ± 168	9883 ± 559
Makespan (s)	9436 ± 122	12090 ± 280	13395 ± 193	11787 ± 186	12436 ± 837

(c) High-level Overheads in Context, Source Data

Figure 5.15: **Performance Breakdown: hmmsearch (fine grain), 343 proteins, two heterogeneous nodes.** Smaller numbers are better. Placeholder overheads are large due to the fine-grain `hmmsearch` jobs. NFS dominates the other strategies. The IMPL strategies beat the EXPL ones, and the SAMBA strategies beat the TFS ones.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).

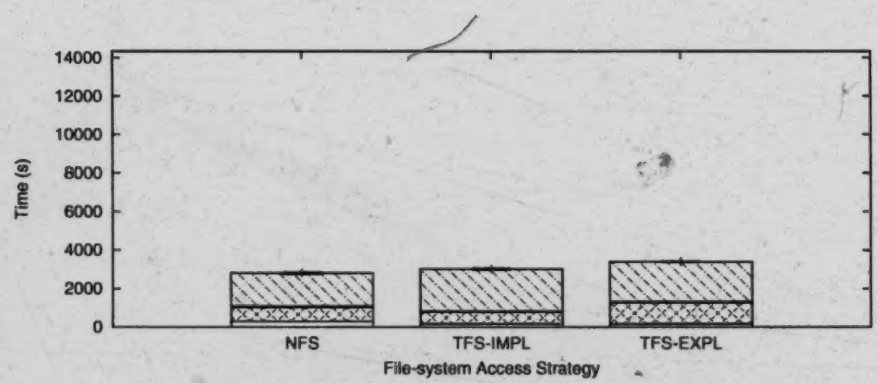


(e) Placeholder Overhead Detail, Normalized

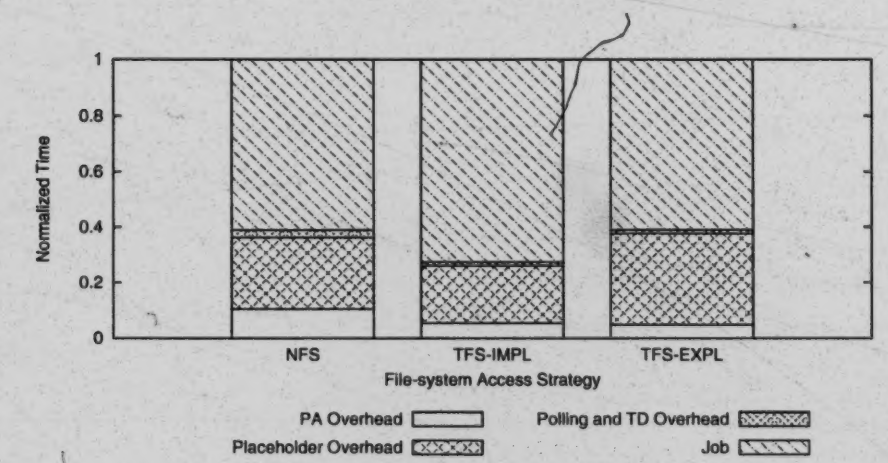
	NFS	TFS-IMPL	TFS-EXPL	SAMBA-IMPL	SAMBA-EXPL
Get Job (s)	851 ± 40	740 ± 38	762 ± 26	756 ± 35	793 ± 100
Notify Job Complete (s)	725 ± 33	645 ± 42	717 ± 30	686 ± 28	765 ± 109
Commit Changes (s)	0 ± 0	0 ± 0	963 ± 60	0 ± 0	1220 ± 146
Notify Commit Complete (s)	0 ± 0	0 ± 0	638 ± 23	0 ± 0	703 ± 82
Placeholder Script Overheads (s)	72 ± 1	75 ± 0	119 ± 1	76 ± 1	119 ± 7
Staggered Placeholder End Times (s)	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0
Placeholder Correction (s)	-78 ± 14	-58 ± 13	-66 ± 9	-56 ± 15	-59 ± 14
Total (s)	1570 ± 71	1404 ± 80	3133 ± 128	1462 ± 57	3542 ± 426

(f) Placeholder Overhead Detail, Source Data

Figure 5.16: **Placeholder Overheads: hmmsearch (fine grain), 343 proteins, two heterogeneous nodes.** Smaller numbers are better. All strategies spend a large amount of time requesting work from the scheduler and signaling job completion. The explicit write-back strategies additionally include a phase to copy output data to the home node (commit changes), which other strategies include in job execution, and an extra message per job to signal completion of the commit step (notify commit complete). Compared with Figure 5.12 (`hmmpfam`, different scale), placeholder overheads are much larger.



(a) High-level Overheads in Context. Legend shared with Figure (b).

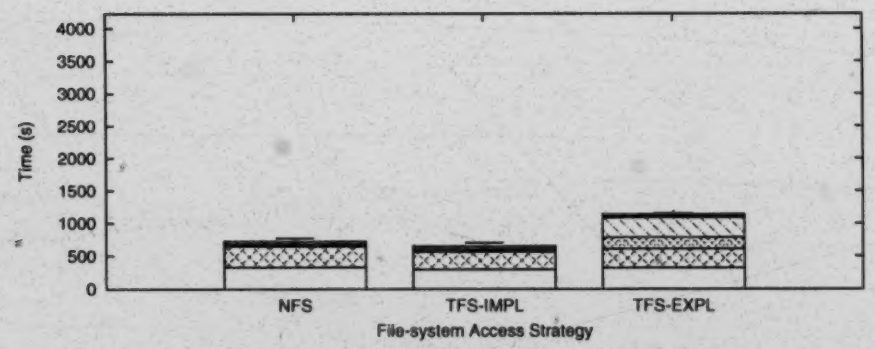


(b) High-level Overheads in Context, Normalized

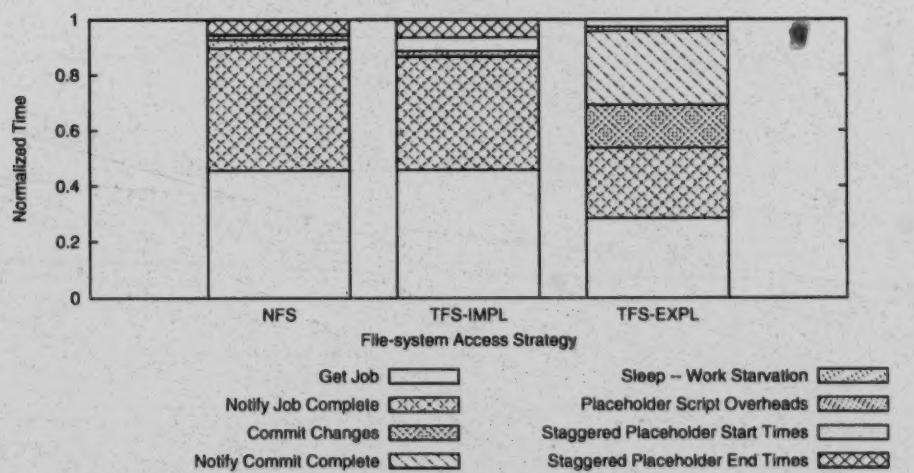
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	5	5	5
PA Overhead (s)	294 ± 4	164 ± 2	164 ± 1
Placeholder Overhead (s)	721 ± 50	623 ± 37	1114 ± 15
Polling and TD Overhead (s)	78 ± 21	39 ± 10	47 ± 12
Job (s)	1715 ± 5	2188 ± 11	2069 ± 5
Job + Commit (s)	1715 ± 5	2188 ± 11	2245 ± 4
Makespan (s)	2808 ± 48	3015 ± 37	3394 ± 17

(c) High-level Overheads in Context, Source Data

Figure 5.17: **Performance Breakdown: hmmsearch (fine grain), 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. With the fine-grain `hmmsearch` jobs, overheads are a major portion of overall makespan. TFS-IMPL is slower than NFS because it copies small files with `scp`, impacting job time, and TFS-EXPL is slower yet because of the commit notification overhead (see Figure 5.18). Some of the increase in placeholder overheads from TFS-IMPL to TFS-EXPL is countered by a reduction in job time due to the different phase in which data is copied to the home node.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).



(e) Placeholder Overhead Detail, Normalized

	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	328 ± 7	299 ± 14	322 ± 9
Notify Job Complete (s)	318 ± 6	266 ± 17	288 ± 3
Commit Changes (s)	0 ± 0	0 ± 0	176 ± 5
Notify Commit Complete (s)	0 ± 0	0 ± 0	302 ± 8
Sleep – Work Starvation (s)	22 ± 15	2 ± 3	0 ± 0
Placeholder Script Overheads (s)	12 ± 1	12 ± 1	19 ± 1
Staggered Placeholder Start Times (s)	2 ± 0	32 ± 7	28 ± 8
Staggered Placeholder End Times (s)	39 ± 30	43 ± 67	1 ± 0
Placeholder Correction (s)	-0 ± 0	-29 ± 14	-22 ± 6
Total (s)	721 ± 50	623 ± 37	1114 ± 15

(f) Placeholder Overhead Detail, Source Data

Figure 5.18: **Placeholder Overheads: hmmsearch (fine grain), 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. As in Figure 5.16, the TFS-EXPL strategy has the extra commit changes and notify commit complete overheads.

5.5.3 Secure-shell Connection Sharing

One source of overheads for the fine-grain `hmmsearch` workloads is the large number of `ssh` connections made. These overheads can be reduced with secure-shell connection sharing. An `ssh` connection is made in the following scenarios:

1. Each time a placeholder requests a job from the scheduler or notifies the scheduler that a job is complete, it invokes `ssh` to connect to the scheduler and execute a script. Each of these invocations results in a new `ssh` connection. In addition, the explicit write-back strategies use the same mechanism to send a message to the scheduler whenever a job completes the commit step.

2. Each time the TFS library launches its RPC agent on the home node, a new `ssh` connection is made. The RPC agent is required for file-system metadata operations (e.g., using `stat()` to ensure a file exists before copying it with `scp`). With the NFS strategy, the RPC agent is never used. With the TFS-IMPL and TFS-EXPL strategies, an RPC agent is launched once per invocation of `hmmpfam` or `hmmsearch` (e.g., twice per job when the batching factor is two). With the SAMBA-IMPL and SAMBA-EXPL strategies, the RPC agent is launched once when Trellis Samba first services a request.

3. For all of the strategies other than NFS, each time an input or output file is copied to or from the home node, `scp` is used. Each invocation of `scp` results in a new `ssh` connection.

With secure-shell connection sharing, `ssh` connections are held open and reused, reducing the latency of these three operations. Figure 5.19 summarizes the impact of connection sharing when using `hmmsearch` with 343 proteins on the four-node cluster configuration; for details, compare Figure 5.20 with Figure 5.17 and Figure 5.21 with Figure 5.18. All strategies benefit from connection sharing, with strategies that initiate more `ssh` connections benefiting more. Note that even the NFS strategy, which uses neither redundancy nor the Trellis File System, benefits slightly from connection sharing due to a reduction in scheduling overheads. Earlier, we saw that TFS-EXPL was 21% slower than NFS with connection sharing in this scenario; without connection sharing, this increases to 33%, i.e. 3863 $\pm$ 31 s for TFS-EXPL versus 2911 $\pm$ 54 s for NFS in Figure 5.20c.

5.5.4 Batching

Our experiments show that batching is an effective way to reduce the overheads introduced by fine-grain workloads. Figure 5.22 shows the impact of batching when `hmmsearch` is used to analyze 343 proteins on the four-node cluster configuration. Increased batching increases job granularity, reduces placeholder overheads, and reduces overall makespan for all strategies. As for connection sharing (Section 5.5.3), even the NFS strategy benefits from batching due to a reduction in scheduling overheads. With the higher batching factor, TFS-EXPL is able to outperform TFS-IMPL due

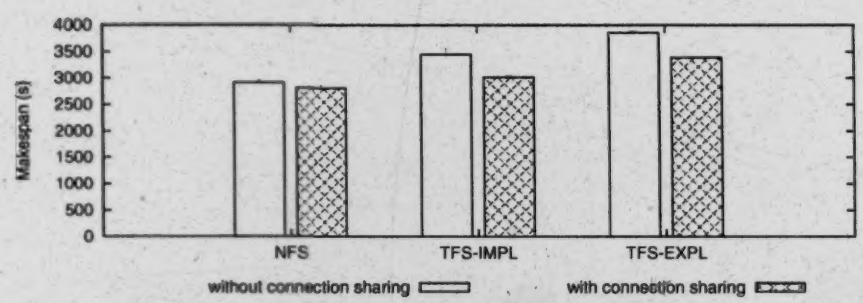


Figure 5.19: **Impact of Secure-shell Connection Sharing: hmmsearch, 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. Connection sharing reduces overheads and improves makespan across the board. See Figures 5.20 and 5.17 for details.

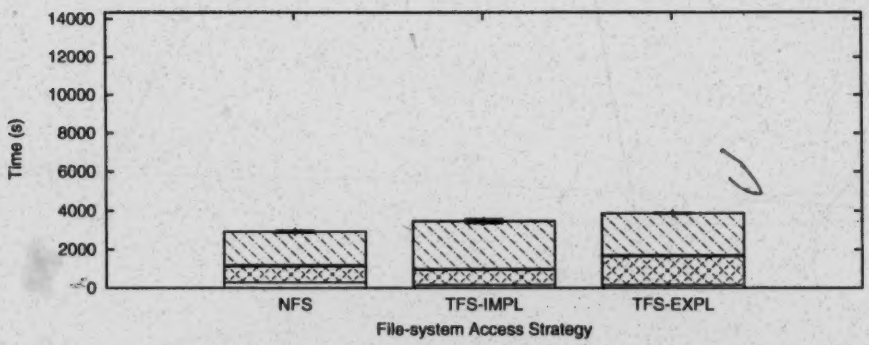
to increased opportunity for caching (each `hmmsearch` execution reuses the same input file) and reduced overheads (due to increased job granularity). One limitation of batching with `hmmsearch` in the current system is that there is one `scp` to the home node – the same node as the scheduler – for every `hmmsearch` invocation. Without batching these as well, the separate `scp` invocations may become a bottleneck when there are more placeholders.

5.6 Future Work: Alternative Workloads

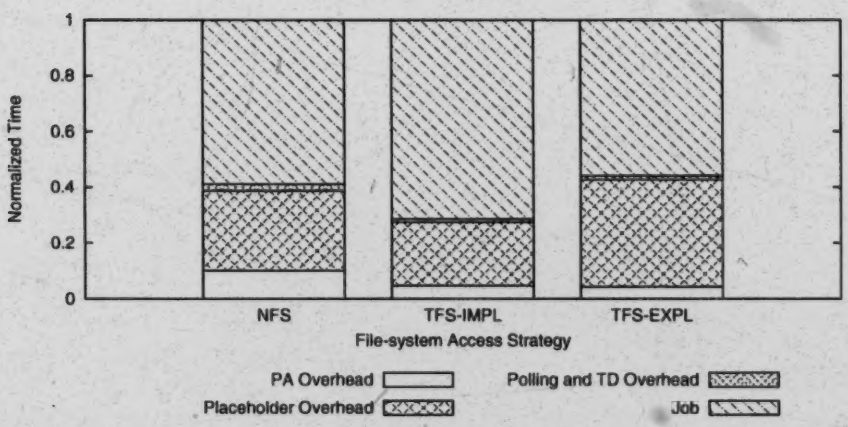
As noted earlier, our evaluation is restricted to high-performance-computing workloads comprised of independent, CPU-dominated jobs that communicate via the file system. Part of this focus derives from Trellis: Executing jobs can not communicate directly with each other, and therefore must access their input and output data via some intermediary (e.g., a distributed file system or a database). Further, our support for redundant execution requires that this intermediary be the Trellis File System. Nonetheless, there are some classes of relevant workloads that we have not explored in our application benchmark. For a subset of these, we speculate what the impact of redundant execution via explicit write-back may be:

1. **Wide Area Networks.** If the application benchmarks were executed in a WAN context, we expect that the latencies of remote operations (e.g., metadata requests) would be higher. To the extent that jobs trigger redundant metadata requests, the explicit write-back strategies would gain an advantage from aggressive caching. We also expect variation in job-completion times to be larger in a WAN context, due to increased heterogeneity and dependence on the network. If this variation is large enough, it might also give an advantage to the explicit write-back strategies, which can mitigate the effect of very slow job executions in the tail.

2. **Dependencies.** In the PA workloads, all jobs are independent. Therefore, it is only at the end of the workload where job-completion latency is more important than job-completion



(a) High-level Overheads in Context. Legend shared with Figure (b).

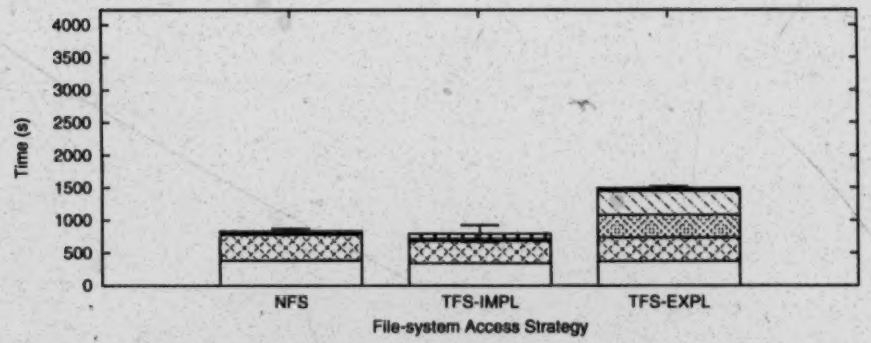


(b) High-level Overheads in Context, Normalized

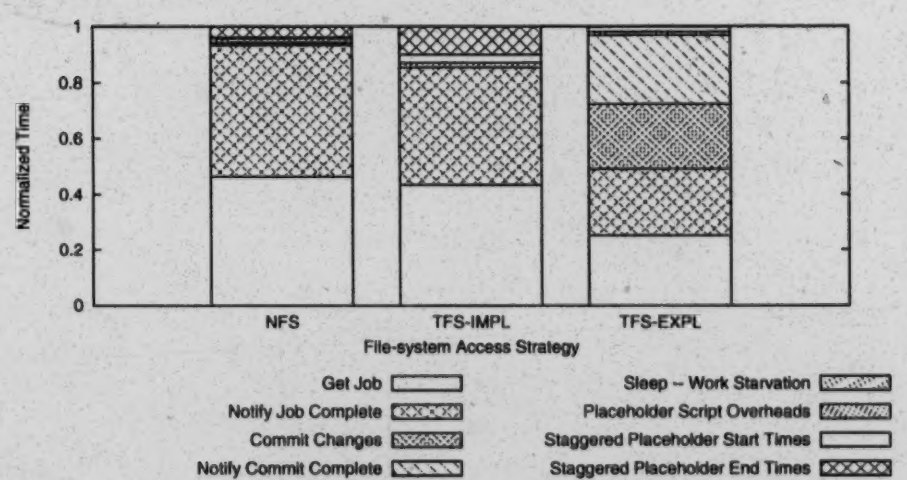
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	6	5	5
PA Overhead (s)	295 ± 14	164 ± 1	166 ± 2
Placeholder Overhead (s)	830 ± 40	785 ± 128	1481 ± 19
Polling and TD Overhead (s)	70 ± 17	41 ± 6	55 ± 13
Job (s)	1715 ± 5	2464 ± 24	2161 ± 4
Job + Commit (s)	1715 ± 5	2464 ± 24	2508 ± 5
Makespan (s)	2911 ± 54	3453 ± 105	3863 ± 31

(c) High-level Overheads in Context, Source Data

Figure 5.20: **Performance Breakdown: hmmsearch (fine grain), 343 proteins, 4×2 cluster CPUs, without secure-shell connection sharing.** Smaller numbers are better. In contrast to Figure 5.17 (with secure-shell connection sharing), all strategies perform worse. This is most evident for placeholder overheads (each message to the scheduler has its own `ssh` connection) and job time for non-NFS strategies (each file transfer and RPC agent launch has its own `ssh` connection).



(d) Placeholder Overhead Detail. Legend shared with Figure (e).



(e) Placeholder Overhead Detail, Normalized

	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	386 ± 5	345 ± 28	374 ± 12
Notify Job Complete (s)	390 ± 9	335 ± 25	360 ± 7
Commit Changes (s)	0 ± 0	0 ± 0	346 ± 4
Notify Commit Complete (s)	0 ± 0	0 ± 0	366 ± 11
Sleep – Work Starvation (s)	9 ± 9	0 ± 0	0 ± 0
Placeholder Script Overheads (s)	12 ± 1	13 ± 1	19 ± 2
Staggered Placeholder Start Times (s)	3 ± 0	23 ± 6	27 ± 7
Staggered Placeholder End Times (s)	32 ± 25	80 ± 175	2 ± 1
Placeholder Correction (s)	-1 ± 0	-11 ± 9	-12 ± 12
Total (s)	830 ± 40	785 ± 128	1481 ± 19

(f) Placeholder Overhead Detail, Source Data

Figure 5.21: **Placeholder Overheads: hmmsearch (fine grain), 343 proteins, 4×2 cluster CPUs, without secure-shell connection sharing.** Smaller numbers are better. In contrast to Figure 5.18 (with secure-shell connection sharing), overheads are larger.

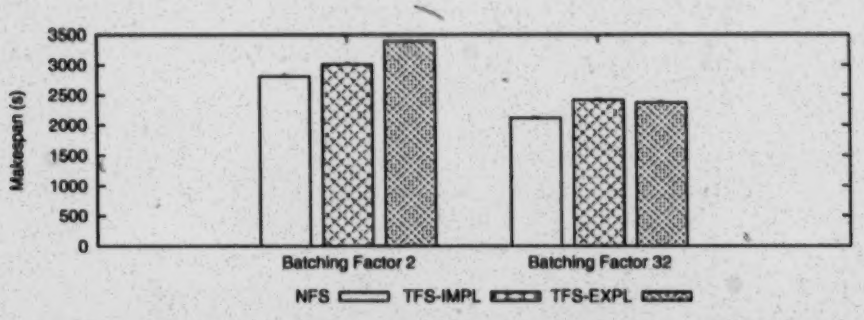


Figure 5.22: **Impact of Batching: hmmsearch, 343 proteins, 4×2 cluster CPUs.** Smaller numbers are better. See Figures 5.17 and 5.23 for details. In this scenario, increased batching significantly reduces makespan for all strategies. With a batching factor of 2, TFS-IMPL outperforms TFS-EXPL; with a batching factor of 32, TFS-EXPL outperforms TFS-IMPL.

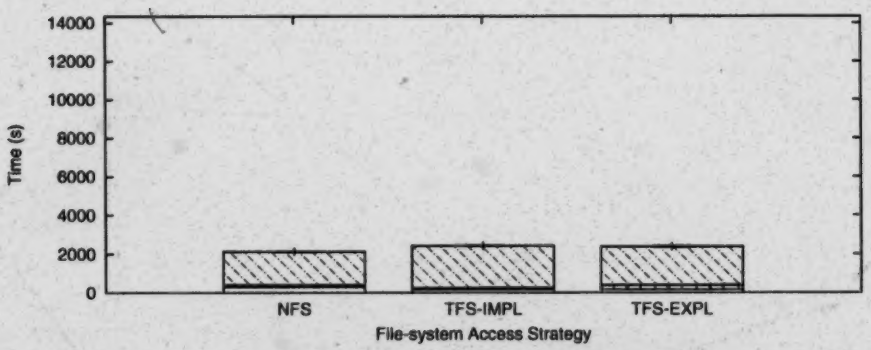
throughput. However, for workloads with dependencies between jobs, there is an opportunity for the explicit write-back strategies to improve makespan by ensuring jobs with dependencies do not take excessively long to complete. However, one limitation of the current implementation is that redundant executions of jobs are not terminated when one of them completes. This means that placeholders might be redundantly executing a completed job after new jobs become available (due to satisfied dependencies). It is unclear how much of a net benefit or liability redundant execution would be in this scenario.

3. **Large Files.** In the PA workloads, the files read and written are very small. The TFS library synchronously copies whole files to and from the home nodes via `scp`, whether or not explicit write-back is used; with explicit write-back, data transfer to the home node occurs later, but is still synchronous. Whole-file data transfer is efficient from the network perspective, as long as all of the data will be accessed, but does incur large latencies when files are large. In contrast, distributed file systems that access data at the block granularity (e.g., NFS) can better overlap data transfer with computation. Consequently, we expect TFS to perform worse than other distributed file systems for very large files and IO-dominated workloads.

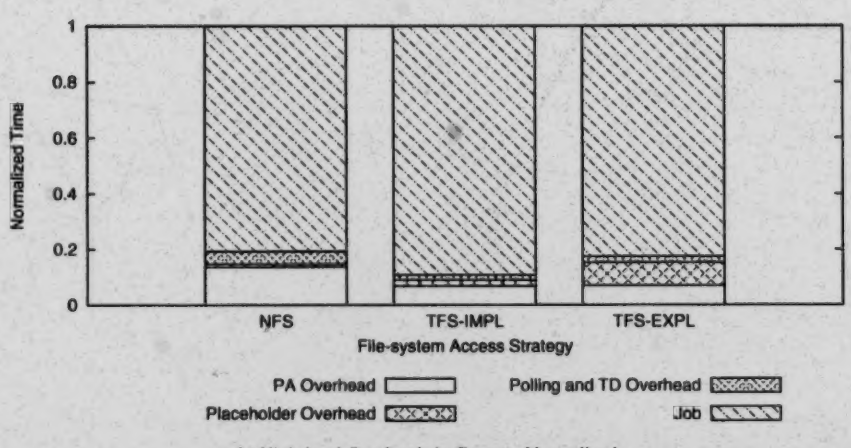
5.7 Concluding Remarks

By benchmarking our system with Proteome Analyst, we have shown that redundant execution allows Trellis to mitigate the effects of slow or failed nodes while introducing negligible overheads for medium- or large-grain CPU-intensive workloads. For fine-grain workloads, we have shown how overheads can be reduced with connection sharing and batching. Specifically:

1. In Section 5.3, we rebooted a placeholder node during workload execution to show that redundant execution allows workloads to complete when placeholders fail. Without redundant execution, the scheduler waits indefinitely for failed placeholders.



(a) High-level Overheads in Context. Legend shared with Figure (b).

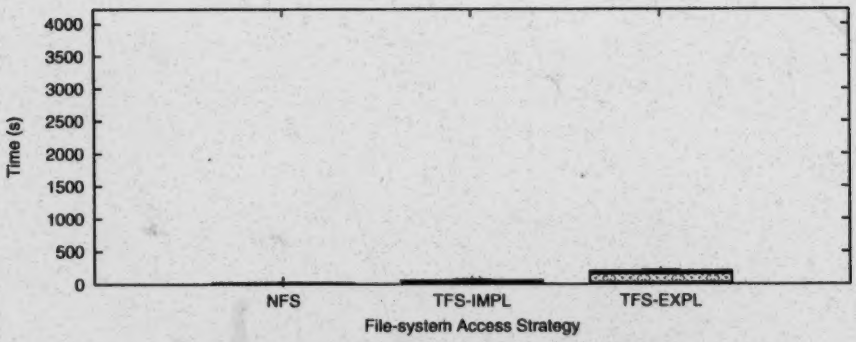


(b) High-level Overheads in Context, Normalized

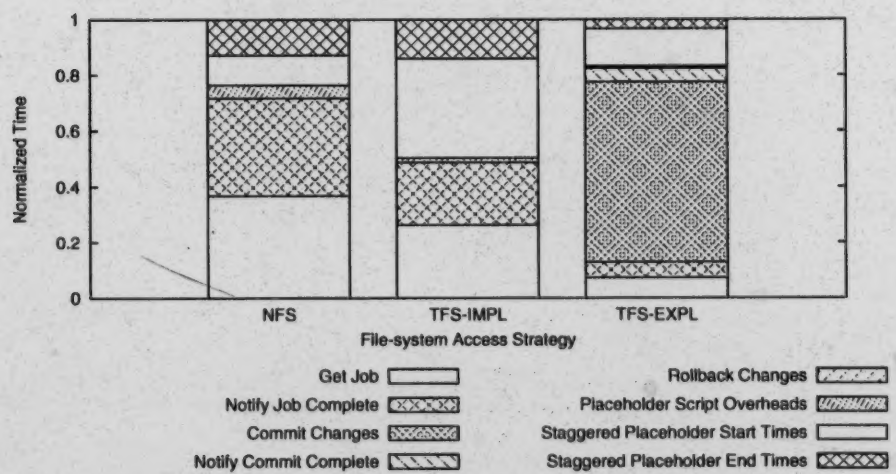
	NFS	TFS-IMPL	TFS-EXPL
Number of Samples	6	6	5
PA Overhead (s)	292 ± 12	165 ± 1	167 ± 2
Placeholder Overhead (s)	22 ± 3	54 ± 9	198 ± 11
Polling and TD Overhead (s)	96 ± 5	42 ± 9	51 ± 6
Job (s)	1712 ± 2	2158 ± 16	1957 ± 33
Job + Commit (s)	1712 ± 2	2158 ± 16	2088 ± 44
Makespan (s)	2121 ± 18	2419 ± 26	2373 ± 38

(c) High-level Overheads in Context, Source Data

Figure 5.23: **Performance Breakdown: hmmsearch (fine grain), 343 proteins, 4×2 cluster CPUs, batching factor 32.** Smaller numbers are better. Compared with Figure 5.17 (batching factor 2), the increased batching factor greatly reduces placeholder overheads, as well as overall makespan. With the larger batching factor, TFS-EXPL is able to outperform TFS-IMPL due to increased opportunity for caching and reduced overheads.



(d) Placeholder Overhead Detail. Legend shared with Figure (e).



(e) Placeholder Overhead Detail, Normalized

	NFS	TFS-IMPL	TFS-EXPL
Get Job (s)	8 ± 2	16 ± 4	15 ± 2
Notify Job Complete (s)	8 ± 1	13 ± 2	11 ± 1
Commit Changes (s)	0 ± 0	0 ± 0	131 ± 12
Notify Commit Complete (s)	0 ± 0	0 ± 0	10 ± 1
Rollback Changes (s)	0 ± 0	0 ± 0	0 ± 0
Placeholder Script Overheads (s)	1 ± 0	1 ± 0	2 ± 1
Staggered Placeholder Start Times (s)	2 ± 0	21 ± 9	27 ± 3
Staggered Placeholder End Times (s)	3 ± 1	8 ± 2	7 ± 1
Placeholder Correction (s)	-0 ± 0	-6 ± 4	-4 ± 3
Total (s)	22 ± 3	54 ± 9	198 ± 11

(f) Placeholder Overhead Detail, Source Data

Figure 5.24: **Placeholder Overheads: hmmsearch (fine grain), 343 proteins, 4×2 cluster CPUs, batching factor 32.** Smaller numbers are better. Compared with Figure 5.18 (batching factor 2), placeholder overheads are greatly reduced. The large commit changes overhead for TFS-EXPL corresponds to the 32 output files copied to the home node for every job.

2. In Section 5.4, we artificially slowed one of four cluster nodes by five or ten times to show that redundant execution provides a performance benefit when some nodes are slow. With redundant execution, the scheduler does not need to wait for slow nodes at the end of the workload.

3. In Section 5.5.1, we used `hmmpfam` to demonstrate that for medium-grain CPU-intensive workloads, the Trellis system as a whole introduces small overheads ($\leq$2%), and the choice of file system, support for redundant execution, or indirection via Trellis Samba makes no significant difference to performance.

4. In Section 5.5.2, we used `hmmsearch` to demonstrate that for fine-grain CPU-intensive workloads, overheads are much larger than for the medium-grain workloads. The fine-grain workloads expose performance differences between the file-system access strategies which are not visible for the medium-grain workloads: (a) NFS is the fastest overall (with some caveats), (b) it is faster to use Trellis Samba than the TFS library directly, and (c) support for redundant execution introduces new overheads.

5. In Section 5.5.3, we showed that `ssh` connection overheads are significant, and demonstrated how secure-shell connection sharing can help reduce them.

6. In Section 5.5.4, we showed that batching jobs together can dramatically reduce overheads for fine-grain workloads.

Chapter 6

Conclusion

As researchers look to scientific computing to answer increasingly difficult questions, the demand for computational resources grows. For batch workloads this means larger clusters, metacomputers, or other aggregations of resources. As these systems become larger, the frequency of localized failure or performance degradation increases. Left unchecked, failure and slowness can dramatically increase makespan (e.g., as a Trellis metacomputer waits for the last job in a workload to complete on a pathologically slow placeholder). This is the long tail problem.

We advocate redundant execution as a general solution to the long tail problem. By redundantly executing jobs at the end of a workload, or before a dependency, the scheduler can transparently bypass placeholders which are executing slowly or have silently failed, thereby reducing makespan.

If multiple placeholders are concurrently executing the same job, they must be protected from interfering with each other; otherwise, they may produce corrupt or incorrect output. In our system we provide the necessary isolation via the Trellis File System. Specifically, we extend TFS with a modular interface for multiple cache policies, including the explicit write-back policy which defers persisting file-system modifications until explicitly instructed by the scheduling system. The scheduler leverages this policy to ensure that concurrent executions of the same job defer persisting their output until one execution successfully completes. Then, this successful execution persists its output and the other executions discard theirs.

We introduce two new ways for user applications to interface with the Trellis File System: Trellis Samba and the TFS Wrapper. Trellis Samba allows unmodified applications to access TFS via a CIFS file-system mount. The TFS Wrapper allows unmodified applications to access TFS via the standard input and output streams; the input stream is populated from an input file, and the output stream is redirected to an output file.

To evaluate the performance impact of redundant execution, we benchmark application workloads from Proteome Analyst under various conditions. Redundant execution introduces negligible overheads for medium- or large-grain CPU-intensive workloads, and enables the system to transparently recover from failed placeholders and avoid waiting for slow ones. Ultimately, this leads to improved performance and reduced administrative burden.

In summary, as discussed in Chapter 1, this thesis makes the following contributions:

1. **Extensions to the Trellis File System.** The TFS library is extended to support the explicit write-back policy, providing a transaction-style interface to the file system. We introduce Trellis Samba and the TFS Wrapper, which provide new ways for user applications to access TFS.

2. **Extensions to the Placeholder Scheduling System.** The scheduler leverages TFS' explicit write-back policy to execute jobs in two phases, first executing jobs redundantly but not persisting their output, and then persisting output for exactly one successful execution of each job.

3. **Quantitative Analysis.** We instrument and benchmark the PA/Trellis system, quantifying the effect of redundant execution on medium- and fine-grain CPU-intensive batch workloads. We also explore the trade-offs between NFS and TFS (accessed directly or via Trellis Samba), along with the impact of secure-shell connection sharing and batching for fine-grain workloads.

Bibliography

[1] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[2] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems (PDCS-96)*, Dijon, France, September 1996.

[3] John Bent, Douglas Thain, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Explicit Control in a Batch Aware Distributed File System. In *Proceedings of the 1st USENIX/ACM Conference on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[4] Nicholas Boers, Alona Fyshe, Paul Nalos, and Adam Rachmielowski. Parallelizing HMMER for Proteome Analyst. Course project, CMPUT 681: Parallel and Distributed Systems, April 2005.

[5] Eugene Brooks. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[6] Michael Closson. The Trellis Network File System. Master's thesis, Department of Computing Science, University of Alberta, 2004.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, California, U.S.A., December 2004.

[8] Robert D. Finn, Jaina Mistry, Benjamin Schuster-Bockler, Sam Griffiths-Jones, Volker Hollich, Timo Lassmann, Simon Moxon, Mhairi Marshall, Ajay Khanna, Richard Durbin, Sean R. Eddy, Erik L. L. Sonnhammer, and Alex Bateman. Pfam: clans, web tools and services. *Nucleic Acids Research*, 34:D247–251, 2006.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, Lake George, NY, October 2003.

[10] Mark Goldenberg, Paul Lu, and Jonathan Schaeffer. TrellisDAG: A System for Structured DAG Scheduling. In *9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, Seattle, Washington, U.S.A., June 2003.

[11] HMMER: profile HMMs for protein sequence analysis. `http://hmmer.wustl.edu/`, 2006.

[12] Nicholas Lamb. Data-Conscious Scheduling of Workflows in Metacomputers. Master's thesis, Department of Computing Science, University of Alberta, 2005.

[13] Nicholas Lamb, Paul Lu, and Alona Fyshe. Trellis Driver: Distributing a Java Workflow Across a Network of Workstations. In *6th International Workshop on High Performance Scientific and Engineering Computing with Applications (HPSEC-04) held with the 33rd International Conference on Parallel Processing (ICPP-04)*, pages 198–205, Montreal, Quebec, Canada, August 2004.

[14] Michael Closson Paul Lu. Bridging Local and Wide Area Networks for Overlay Distributed File Systems. In *2nd USENIX Workshop on Real, Large Distributed Systems (WORLDS '05)*, pages 49–54, San Francisco, California, U.S.A., December 2005.

[15] Paul Lu, Michael Closson, Cam Macdonell, Paul Nalos, Danny Ngo, Morgan Kan, and Mark Lee. The Trellis security infrastructure for overlay metacomputers and bridged distributed file systems. *Journal of Parallel and Distributed Computing*, 66:1181–1188, 2006.

[16] Mycoplasma pneumoniae complete proteome. `http://www.expasy.org/sprot/hamap/MYCPN.html`, 2006.

[17] Michael A. Olson. The Design and Implementation of the Inversion File System. In *Proceedings of the USENIX Winter 1993 Technical Conference*, San Diego, California, U.S.A., January 1993.

[18] Christopher Pinchak. Placeholder Scheduling for Overlay Metacomputing. Master's thesis, Department of Computing Science, University of Alberta, 2002.

[19] Christopher Pinchak, Paul Lu, and Mark Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *8th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 85–105, Edinburgh, Scotland, U.K., July 2002.

[20] Christopher Pinchak, Paul Lu, Jonathan Schaeffer, and Mark Goldenberg. The Canadian Internetworked Scientific Supercomputer. In *17th International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 193–199, Sherbrooke, Quebec, Canada, May 2003.

[21] Brett Poulin, Duane Szafron, Russell Greiner, Roman Eisner, and Paul Lu. RAMMER: Accelerating Protein Function Prediction. Canadian Proteomics Initiative, 2005.

[22] Sean Rhea, Byung-Gon Chun, John Kubiatowiczi, and Scott Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proceedings of USENIX WORLDS 2005*, pages 25–30, San Francisco, California, U.S.A., December 2005.

[23] Samba. `http://samba.org`, 2006.

[24] Frank Schmuck and Jim Wyllie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 239–253, Pacific Grove, CA, October 1991.

[25] Jeff Siegel and Paul Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (Cluster 2002)*, pages 480–483, Chicago, Illinois, U.S.A., September 2002.

[26] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 3rd Edition.* McGraw-Hill Book Company, 1997.

[27] Paul Stelling, Cheryl DeMatteis, Ian Foster, Carl Kesselman, Craig Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.

[28] Craig Stewart, Christopher Peebles, Mary Papakhian, John Samuel, David Hart, and Stephen Simms. High performance computing: delivering valuable and valued services at colleges and universities. In *SIGUCCS '01: Proceedings of the 29th annual ACM SIGUCCS conference on User services*, pages 266–269, New York, NY, USA, 2001. ACM Press.

[29] Duane Szafron, Paul Lu, Russell Greiner, David S. Wishart, Brett Poulin, Roman Eisner, Zhiyong Lu, John Anvik, Cam Macdonell, Alona Fyshe, and David Meeuwis. Proteome Analyst: Custom Predictions with Explanations in a Web-based Tool for High-throughput Proteome Annotations. *Nucleic Acids Research*, 32:W365–W371, July 2004.

[30] Douglas Thain, John Bent, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Pipeline and Batch Sharing in Grid Workloads. In *Proceedings of the 12th IEEE Symposum on High Performance Distributed Computing*, pages 152–161, Seattle, WA, June 2003.

[31] The Trellis Project. `http://www.cs.ualberta.ca/~paullu/Trellis`, 2006.

[32] Charles Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Amino: Extending ACID Semantics to the File System. In *Work-in-Progress Reports at the FAST '05 Conference on File and Storage Technologies*, San Francisco, California, U.S.A., December 2005.

[33] Gosia Wrzesinska, Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Fault-Tolerant Scheduling of Fine-Grained Tasks in Grid Environments. *International Journal of High Performance Computing Applications*, 20(1):103–114, 2006.

[34] Yunjie Xu, Aiko Huckauf, Wolfgang Jäger, Paul Lu, Jonathan Schaeffer, and Christopher Pinchak. The CISS-1 Experiment: ab initio Study of Chiral Interactions. In *39th International Union of Pure and Applied Chemistry (IUPAC) Congress and 86th Conference of The Canadian Society for Chemistry*, Ottawa, Ontario, Canada, August 2003.

Appendix A

Acronyms and Programs

A.1 List of Acronyms

ACID	atomicity, consistency, isolation, and durability
AMD	Advanced Micro Devices, Inc.
API	application programming interface
BLAST [1]	Basic Local Alignment Search Tool
CIFS	Common Internet File System
CISS [20]	Canadian Internetworked Scientific Supercomputer
CLS [19, 18]	Command Line Server
CPU	central processing unit
DNA	deoxyribonucleic acid
HMMER [11]	a tool which compares protein sequences with hidden Markov models
HMM	hidden Markov model
HPC	high-performance computing
IO	input, output
LAN	local area network
NFS	Network File System
PA/Trellis	Proteome Analyst extended to use Trellis
PA [29]	Proteome Analyst
POSIX	Portable Operating System Interface
RAMMER [21]	an accelerated variant of HMMER
RPC	remote procedure call
SAMBA-EXPL	denotes use of Trellis Samba with the explicit write-back policy
SAMBA-IMPL	denotes use of Trellis Samba with the implicit write-back policy
SCL [25]	secure copy locator
SFTP	Secure File Transfer Protocol
TD	Trellis Driver (used only to save space in figures and tables)
TFS-EXPL	denotes use of the Trellis File System with the explicit write-back policy
TFS-IMPL	denotes use of the Trellis File System with the implicit write-back policy
TFS [25]	Trellis File System
VFS	virtual file system
WAN	wide area network

A.2 List of Programs

`hmmpfam`	part of HMMER; may be used to compare one protein with many HMMs
`hmmsearch`	part of HMMER; may be used to compare one HMM with many proteins
`ssh`	secure shell
`scp`	secure copy
`sshd`	secure shell daemon
`mqsub`	job submission script